MOTEUR D'INFÉRENCE PARALLÈLE POUR TRANSPUTER ALEX

par

Jean-Pierre Grenier

mémoire présenté au Département de mathématiques
et d'informatique en vue de l'obtention du grade de maître ès sciences (M.Sc.)

FACULTÉ DES SCIENCES
UNIVERSITÉ DE SHERBROOKE

Sherbrooke, Québec, Canada, décembre 1996

Sommaire

Le problème abordé dans ce mémoire est la conception et le développement d'un moteur d'inférence pouvant résoudre un problème parallélisé sur un multiprocesseur. Pour cette tâche, nous utilisons un transputer à 16 processeurs et nous établissons un lien de communication entre la machine parallèle et un logiciel d'interface graphique.

Dans ce mémoire, nous présentons une solution à ce problème. Nous avons conçu et développé un modèle de moteur d'inférence parallèle utilisant le paradigme de la communication par messages. Ce modèle de moteur d'inférence à base de règles utilise des *faits* représentés à l'aide de triplets (*objet, attribut, valeur*) et des *règles* représentées par des *antécédents* et des *conséquents*. Nous avons mis au point un langage adapté aux besoins de communication inter-noeuds, ce qui permet au moteur d'inférence d'être générique. Un mécanisme limitant le nombre de messages inter-noeuds le distingue. Le lien avec le logiciel d'interface graphique permet de visualiser l'état des bases de faits réparties ainsi que d'interagir avec celles-ci via des primitives de communication et des ports TCP/IP. Nous présentons deux exemples relevant du domaine du contrôle et développés à l'aide du moteur d'inférence. Les deux exemples interagissent avec le logiciel d'interface graphique.

Remerciements

Je tiens à remercier mon directeur de recherche, M. Michel Barbeau, qui, par son sens de l'organisation et son professionalisme, a été une véritable inspiration. Je le remercie pour l'aide financière qu'il m'a accordée afin de poursuivre mes études. Je remercie également mon co-directeur de recherche, M. Richard St-Denis, pour ses nombreux conseils et ses critiques, toujours positives, de ce mémoire. Travailler avec eux fut, à la fois, un plaisir et un enrichissement.

Je tiens aussi à remercier M. Jean-Marc Palmier, pour son souci du détail dans la conception des deux interfaces qu'il a créées pour les exemples qui accompagnent ce mémoire.

Finalement, je remercie ma soeur Nicole pour le courage dont elle a fait preuve dans ses efforts pour me faire saisir les multiples subtilités de la langue française.

Table des matières

Liste des tableaux

Liste des figures

Introduction

Ce mémoire porte sur les moteurs d'inférence à base de règles de production. Dans cette introduction, nous survolons quelques concepts élémentaires des moteurs d'inférence et nous introduisons la problématique particulière faisant l'objet de ce mémoire. Puis, nous décrivons la méthodologie utilisée et nous esquissons les résultats obtenus. Finalement, nous présentons l'organisation de ce mémoire.

Les moteurs d'inférence

Dans sa démarche pour reproduire la pensée humaine, l'homme a conçu différents mécanismes de raisonnement [Cho85]. Le *moteur d'inférence* est un de ces mécanismes. Mais qu'est-ce qu'un moteur d'inférence? Par définition, un **moteur** est une *cause d'action* [Lar94] et une **inférence** est une *opération par laquelle on passe d'une vérité à une autre* [Lar94].

En associant ces définitions figuratives à notre propos, les *vérités* d'une inférence sont représentées par des *faits* et des *règles de production.* Les *opérations*, effectuées par un moteur, consistent en des *phases d'appariement*, de *résolution des conflits* puis d'*exécution d'une règle.*

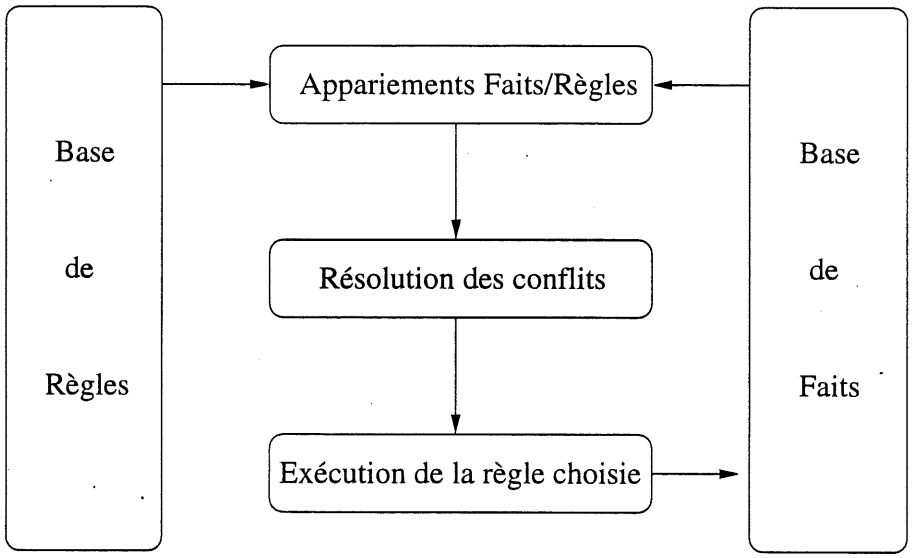


Figure 1: Moteur d'inférence type

Nous pouvons désormais présenter succinctement les concepts élémentaires d'un moteur d'inférence à base de règles de production. Comme l'illustre le modèle de la figure 1, un moteur d'inférence agit sur une base de faits à l'aide d'une base de règles. Le moteur possède un cycle d'évaluation constitué de trois étapes: l'*appariement*, la *résolution des conflits* et l'*exécution d'une règle*. À l'appariement, chaque fait est comparé aux différents éléments des règles de production pour déterminer s'il y a compatibilité entre les connaissances connues (les faits) et les prérequis (les *antécédents* d'une règle), et ainsi permettre l'acquisition de nouvelles connaissances. Si après appariement, plusieurs règles sont satisfaisables, c'est-à-dire que, pour chacune de ces règles, tous les antécédents sont satisfaits, alors il y a résolution de conflits (ou sélection de la règle à exécuter). À cette étape, une règle est choisie à l'aide d'une heuristique. Finalement, l'exécution de la règle consiste à évaluer ses *conséquents*, ce qui implique généralement l'exécution de certaines fonctions et la modification de la base de faits. Ce cycle se répète tant et aussi longtemps qu'une condition d'arrêt n'est pas satisfaite.

Le problème

Ce projet consiste à concevoir et développer un moteur d'inférence pouvant résoudre un problème parallélisé sur un multiprocesseur. La machine parallèle ALEX [Ale92a] que nous utilisons est en fait un *transputer* à 16 processeurs sur lequel s'exécutent les moteurs d'inférence. Nous devons aussi établir un lien de communication entre la machine parallèle ALEX et un logiciel d'interface graphique appelé VAPS [Vir93]. VAPS est un acronyme pour Virtual Avionics Prototype System.

La parallélisation d'un problème consiste à morceler l'ensemble des règles de production et l'ensemble des faits, à les regrouper par objets et à les redistribuer sur les différents processeurs afin que chacun d'eux soit responsable de la gestion de certains objets. Nous limiterons notre recherche aux premiers aspects mentionnés, soit le moteur d'inférence et le lien avec le logiciel d'interface graphique.

Chaque moteur d'inférence réagit à trois sources de stimuli : les nouveaux faits qu'il peut produire par ses propres inférences, les nouveaux faits qu'il peut recevoir des autres processeurs et les événements que l'utilisateur peut produire à l'aide de l'interface graphique de VAPS. La figure 2 illustre ces différentes sources de stimuli. Dans la figure 2, les flèches indiquent les influences que peut subir la base de faits d'un processeur ; il va de soi que ces relations sont symétriques à tous les processeurs.

La méthodologie

Nous avons utilisé une méthode traditionnelle d'analyse et conception, soit l'élaboration de la structure du déroulement d'un programme. Nous avons déterminé les grandes étapes à suivre pour permettre une inférence : soit la lecture des fichiers, l'appariement, la résolution des conflits, l'exécution d'une règle et finalement la communication. Nous

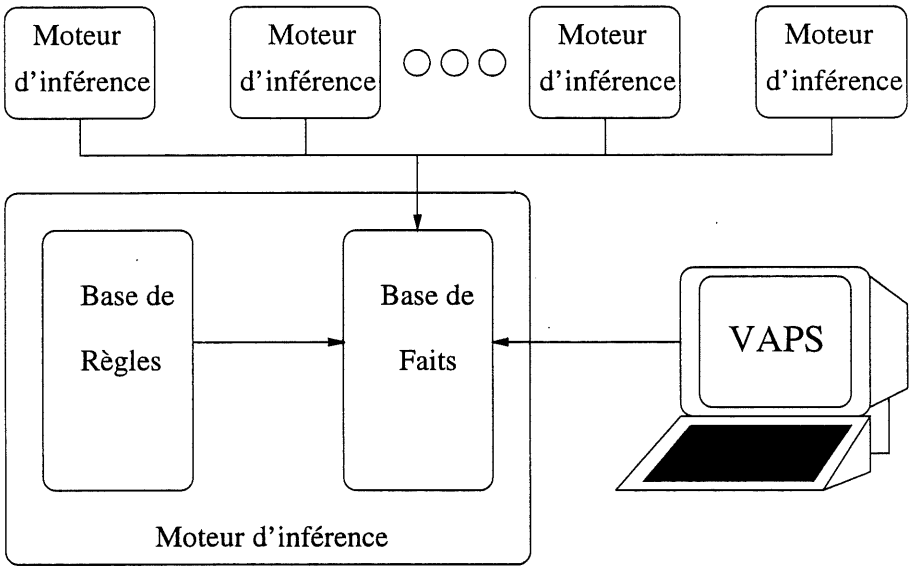


Figure 2: Divers stimuli d'un moteur d'inférence parallèle

avons développé des procédures appropriées à chaque étape.

Pour l'implantation, nous avons opté pour une méthode fonctionnelle : on accède à chaque structure de données à l'aide de constructeurs, de sélecteurs et de mutateurs. Chaque procédure est regroupée dans une famille de procédures et un fichier approprié lui est affecté.

Les résultats

Notre moteur d'inférence parallèle démontre que le paradigme de la communication par messages est un moyen adéquat de gestion des inférences distribuées. Il présente aussi un modèle de contrôle des messages basé sur une association *objet, attribut, processeur.* Cette association permet de limiter le nombre de messages inter-processeurs au minimum dû au fait qu'un processeur ne reçoit que les *faits* qui lui sont pertinents. L'utilisation d'une représentation interne uniquement à base d'entiers permet d'optimiser les routines d'accès aux différents champs des tuples et d'obtenir ainsi un code performant. Outre

l'utilisation exclusive d'entiers dans sa représentation interne, notre modèle présente aussi l'originalité de pouvoir communiquer avec un logiciel d'interface graphique s'exécutant sur une machine différente à l'aide du protocole de communication TCP/IP. Le succès de l'interconnection du logiciel VAPS et de la machine parallèle ALEX est en soi un résultat intéressant.

Organisation du mémoire

Le chapitre 1 présente un aperçu des notions de base relatives aux moteurs d'inférence, soit : les types de connaissances et leur représentation, les composants d'une règle, l'ensemble des conflits et sa résolution, les différents types de parallélisme et finalement la communication. Le chapitre 2 introduit notre modèle, son fonctionnement et la communication avec VAPS. Le chapitre 3 présente deux exemples d'utilisation du moteur et les résultats obtenus, démontrant ainsi sa généralité. Une conclusion suit avec un rappel sur le travail effectué et des suggestions de recherches futures.

Chapitre 1

Eléments de base

Dans ce chapitre, nous présentons dans une première section la terminologie particulière aux moteurs d'inférence à base de règles de production et dans une seconde section les concepts élémentaires du parallélisme.

Comme nous l'avons déjà mentionné, un moteur d'inférence à base de règles de production est un mécanisme permettant de passer d'une connaissance à une autre en imitant un type de pensée humaine. Nous allons donc reprendre quelques éléments de cette affirmation pour en définir le sens et en comprendre les implications.

1.1 Moteur d'inférence à base de règles de production

Dans cette section, nous présentons les concepts de base relatifs aux moteurs d'inférences à base de règles de production, soit : les *types de connaissances et leurs représentations*, les notions d'*antécédents* et les notions de *conséquents*. Nous définissons aussi la notion d'*inférence* et présentons le concept d'*ensemble des conflits*.

1.1.1 Types de connaissances

Établissons d'abord certains *types de connaissances* pour comprendre le domaine d'application propre aux moteurs d'inférence. Nous verrons à la section 2.4 quels types de connaissances notre modèle reconnaît. Les orientations de recherche future proposées en conclusion font référence à certains éléments présentés ici. Nous allons énumérer quelques types de connaissances accompagnés d'exemples. Les notes d'un cours de deuxième cycle de St-Denis [StD94] servent de base à l'élaboration de cette liste.

Vocabulaire : On retrouve ici les différents termes du discours, tous les mots connus sans tenir compte de leur utilisation ou contexte.

Objet : Les objets représentent généralement des entités du monde réel ou des notions complètes en elles-mêmes : une *table* représente un objet réel, un *cours d'informatique* peut être considéré comme un objet. Les objets sont représentés par des noms.

Relation entre les objets : La chaise est *près* de la table. Une relation de *proximité* est établie entre la chaise et la table. Ce cours *requiert* beaucoup d'études. Une relation de *nécessité* unit le cours aux études. Les relations sont souvent nommées par des prépositions.

But : La notion d'achèvement "*obtenir* un diplôme", implique une fin, tout comme "le transfert des données s'est *effectué* sans erreur". Un verbe, conjugué ou non, représente cette notion de but.

Fait : Souvent représenté par l'association *objet, attribut, valeur*, un fait réflète une idée complète :

$$\text{La } \overbrace{\text{table}}^{objet} \overbrace{\text{est de couleur}}^{attribut} \overbrace{\text{rouge.}}^{valeur}$$

L'objet représente quelque chose de réel, l'attribut est une caractéristique de l'objet et finalement la valeur indique l'état de l'attribut.

Hypothèse : Une hypothèse est une supposition utilisée pour démontrer de nouveaux faits. Les hypothèses sont souvent utilisées comme base de raisonnement.

Contrainte : "il *n'y a que* cinq places dans cette automobile" et "cet interrupteur *doit* être activé *avant* celui-ci" sont des exemples de contraintes. Une contrainte représente une limite ou un ordonnancement.

Processus : Un processus est une tâche qui s'exécute. Un programme calculant les facteurs d'un nombre est un processus. La réfection de l'autoroute est aussi un processus. Un processus implique la notion de changement.

Heuristique : Les heuristiques sont des méthodes, plus ou moins justifiées, nous permettant de décider quelque chose. Les heuristiques de décision ont souvent une consonance proverbiale : "il mange le gâteau avant le glaçage, car il aime *garder le meilleur pour la fin*"

Aspect Temporel : Les aspects temporels d'une situation regroupent souvent la notion de contrainte vue précédemment : "le train quitte la gare à 16:00" ou "vous disposez de deux semaines pour faire ce travail" sont deux exemples d'aspect temporel.

Action : "*acheter* son billet pour la partie de hockey" et "*tondre* la pelouse" sont des actions facilement reconnaissables par l'utilisation des verbes d'action.

Événement : Des connaissances passées telles que "il y *avait* trois personnes à la réunion" et "la tornade *a fait* huit morts et quarante blessés" sont des exemples d'événements. Les verbes d'événement sont conjugués au passé.

Règle de décision : "l'entrée est interdite aux enfants de moins de six ans" est un exemple de règle de décision. On peut qualifier les règles de décision de contraintes actives.

Causalité : "tout travail mérite salaire" et "c'est grâce à ses efforts qu'il a réussi" sont des exemples de causalité.

Motivation : "elle *désire* devenir astronaute" et "s'il *continue* à pratiquer la clarinette, il pourra participer au Festival des Harmonies" sont des éléments de motivation. La notion de motivation implique le désir d'atteindre un but dans un futur plus ou moins rapproché.

Comportement : "il aime bien aider les gens" et "cet enfant ment souvent" sont des exemples de comportement. Les comportements sont des choix inconscients alors que les heuristiques sont des décisions d'actions conscientes.

Cause : Dans "les pluies abondantes des derniers jours ont ravagé son jardin", *les pluies* sont la cause du *ravage*.

Cette énumération est non-exhaustive et sujette à controverse. Argumenter à son sujet ne relève pas de notre propos, mais son énumération révèle que ce champ d'études est vaste et varié.

1.1.2 La représentation des connaissances

Pour son fonctionnement, un moteur d'inférence doit disposer d'un mécanisme d'abstraction lui permettant de représenter les connaissances du monde réel en données traitables par un ordinateur. Nous savons que les mécanismes d'abstraction sont reliés de très près à la représentation des données. Nous savons aussi que les mécanismes d'abstraction doivent être à même de fournir une traduction du monde réel avec une granularité suffisamment fine pour permettre plusieurs connexions sur un ensemble de données. Supposons, que nous ayons la connaissance suivante :

$$\overbrace{\text{Le chat de Myra est gris.}}^{A}$$

Si cette information est emmagasinée comme un ensemble monolithique, le moteur ne pourra déduire rien d'autre que : *Le chat de Myra est gris.* Nous serons en présence d'un moteur de type *calcul de pròposition.* Les règles de proposition d'un tel moteur d'inférence sont du genre:

A implique B

A implique C

B et D impliquent E

Par contre, si la connaissance est traduite et conservée sous forme de sous-ensembles distincts :

$$\overbrace{\text{Le chat}}^{A}\ \overbrace{\text{de Myra}}^{B}\ \overbrace{\text{est gris}}^{C}$$

Le moteur d'inférence sera du type *calcul de prédicat* et utilisera des règles comme:

A(B,C) impliquent D

L'accès à plusieurs champs d'un élément de connaissance permet de répondre à plusieurs questions :

Existe-t-il un chat gris?	$x($y,gris)
À qui appartient le chat gris?	Chat($y,gris)
Myra a-t-elle un animal?	$x(Myra,$y)

Si la structure de données utilisée permet d'accéder indépendamment à chaque élément de la connaissance, il devient possible de construire un moteur plus *puissant* au sens déductif du terme. La puissance d'un moteur est dénotée par son *ordre.* On reconnaît cinq ordres.

Ordre 0 : Aussi connu sous le nom de calcul de proposition. Un moteur d'ordre 0 n'utilise que des constantes dans ses règles.

Ordre 0+ : Un moteur d'ordre 0+ est un moteur d'ordre 0, mais il peut aussi utiliser des pseudo-variables, nommées ainsi, car il s'agit de variables dont l'instanciation est statique et ne change pas à l'exécution.

Ordre 1 : Ou calcul de prédicat du premier ordre.

Ordre 1+ : Un moteur d'ordre 1 avec utilisations de variables pour les prédicats (ou nom de relations).

Ordre 2 : Le calcul de prédicat du deuxième ordre permet l'utilisation de variables où se situent normalement des fonctions.

1.1.3 Les antécédents

Une base de règles de production est un ensemble de formules composées d'une conjonction d'antécédents et d'une conjonction de conséquents tel que nous montre la formule suivante :

$$\overbrace{a_1 \wedge \cdots \wedge a_n}^{antécédents} -> \overbrace{c_1 \wedge \cdots \wedge c_m}^{conséquents}$$

Voici un exemple plus éloquant permettant de bien saisir les notions d'antécédent et conséquent :

$$Si\ \overbrace{\text{je dois sortir}}^{antécédent}\ et\ \overbrace{\text{qu'il pleut}}^{antécedent},\ alors\ \overbrace{\text{j'apporte un parapluie.}}^{conséquent}$$

Si A et B alors C

Chaque antécédent représente une connaissance ou une action. Une connaissance peut être *simple* ou *composée.*

Connaissance simple : On définit une connaissance simple comme étant une connaissance dont les éléments de base sont atomiques (non quantifiés), ainsi *un chat gris* est une connaissance simple que l'on pourrait représenter par *Chat(Couleur,Gris) ou A(B,C).*

Connaissance composée : Une connaissance composée est formée d'éléments de base quantifiés, par exemple ainsi *un gros chat gris foncé* peut être représenté par *Chat(Taille(Gros),Teinte(Couleur(Gris)) ou A(B(1),C(D(2)))*.

Certains moteurs d'inférence permettent aux composants d'une règle (notamment les antécédents) de contenir des variables. Cette flexibilité augmente la puissance de déduction du moteur. Voici une règle dont un antécédent contient une variable :

Si A($x) et B(3) alors C(4)

Dans cet exemple, le signe de dollar a été utilisé pour indiquer le début du nom d'une variable.

Pour clore cette section sur les antécédents, nous introduirons la définition de *arité*.

L'arité d'un règle : On appelle *arité d'une règle* le nombre d'antécédents que la règle possède.

Certains moteurs utilisent la notion d'arité dans les heuristiques de résolution de conflits.

1.1.4 Les conséquents

Les conséquents d'une règle représentent les actions à poser et les connaissances acquises par la déduction basée sur les antécédents de la règle. Ce sont généralement les conséquents d'une règle qui alimentent le moteur d'inférence en faits nouveaux. Avec ces nouvelles connaissances, le moteur peut satisfaire de nouvelles règles et déduire encore de nouveaux faits. Comme nous l'expliquons à la section 1.1.5 sur les inférences, les conséquents peuvent obtenir dynamiquement une valeur, c'est-à-dire des valeurs déduites selon le contexte et le moment d'inférence.

Prenons comme exemple une compagnie ayant comme politique qu'un employé qui fait très bien son travail mérite une prime calculée en fonction de son salaire et du poste qu'il occupe :

$$\overbrace{\$\text{employé(SALAIRE,\$salaire)}}^{\textit{antécédent}}\ et$$
$$\overbrace{\$\text{employé(POSTE,\$poste)}}^{\textit{antécédent}}\ et$$
$$\overbrace{\$\text{poste(PRIME,\$prime)}}^{\textit{antécédent}}\ alors$$
$$\overbrace{\$\text{employé(PRIME,\$salaire*\$prime)}}^{\textit{conséquent}}$$

Ainsi, cette règle d'affaire s'écrit à l'aide d'une règle à trois antécédents et un conséquent.

1.1.5 Les inférences

La connaissance de toutes les vérités contenues dans les antécédents d'une règle est le préalable à la déduction de nouveaux faits. Cette reconnaissance s'effectue à l'aide d'un mécanisme appelé *appariement.* Il s'agit de l'étape dans l'exécution du moteur où ce dernier compare les structures de données contenant les faits connus et les structures de données contenant les antécédents des règles. Il y a appariement si et seulement si les éléments des deux types de données sont compatibles. La compatibilité entre deux éléments de données est acquise si les deux structures contiennent des valeurs identiques ou si l'une des structures contient une variable, car une variable est unifiable à n'importe quelle valeur. Une contrainte d'intégrité inter-éléments s'impose pour compléter l'appariement et maintenir la cohérence de la règle.

Contrainte d'intégrité inter-éléments : Chaque variable apparaissant dans un antécédent de règle devra être liée à une seule valeur, même si cette variable apparaît dans un autre antécédent ou dans un conséquent.

Le comportement à associer à l'appariement de deux variables distinctes relève de l'implantation. Il n'y a pas de règle canonique à ce sujet. Certains moteurs permettent à deux variables distinctes d'être liées au même élément de connaissance alors qu'avec d'autres, cela n'est pas permis.

Lorsque toutes les variables d'une règle sont liées à des éléments de données, existants pour les antécédents et possiblement nouveaux pour les conséquents, que toutes les constantes d'une règle ont trouvé parité dans les faits et qu'il n'y a pas de conflits inter-éléments, on nomme cette règle *instanciable*. Une règle instanciable est une règle prête à l'inférence. Mais avant de procéder à l'inférence proprement dite, il faut construire l'*ensemble des conflits* et choisir la règle à exécuter parmi cet ensemble.

1.1.6 L'ensemble des conflits

Il arrive parfois, voire même souvent, qu'après avoir vérifié l'instanciation des règles, on obtienne plusieurs règles instanciables. Nous appelons ce groupe de règles *l'ensemble des conflits*. Choisir une règle parmi cet ensemble constitue la *résolution des conflits*. Chaque moteur d'inférence procède par heuristique [du grec *heuriskein*, trouver] pour élire la règle à instancier. Souvent, les heuristiques ont une allure proverbiale. Nous allons énumérer quelques heuristiques parmi les plus communément utilisées afin d'en saisir l'utilité.

Premier arrivé, premier servi : La première règle trouvée est exécutée. Cette heuristique a l'avantage de minimiser le calcul de l'ensemble des conflits, car sitôt une règle trouvée instanciable, elle est choisie. Mais l'heuristique a un désavantage important, elle privilégie l'ordre d'apparition des règles et risque de causer la *famine* de certaines règles. Une règle est dite en famine si, malgré qu'elle soit instanciable, elle n'est jamais exécutée.

La plus grande arité en premier : La règle ayant le plus grand nombre d'antécédents est exécutée. Cette heuristique est basée sur l'adage qui dit que : *ce qui est le plus dur à obtenir doit avoir plus de valeur.* Si tous les antécédents ont le même risque statistique d'être satisfait, alors il est vrai que satisfaire $n+1$ antécédents sera plus difficile que d'en satisfaire n, mais cela n'est que très rarement le cas. Il y a toujours des cas dégénérés pour lesquels une heuristique est erronée.

La règle la plus prioritaire en premier : Certains moteurs admettent des règles ayant des priorités d'exécution. Cette heuristique donne au concepteur des règles plus de souplesse sur le contrôle du déroulement des inférences, mais elle circonscrit le déroulement de l'exécution à un scénario que le concepteur a prévu.

L'âge des faits utilisés : Parfois, la règle instanciable par les faits ayant été inférés le plus récemment, est élue pour être exécutée la prochaine. Plus coûteuse, car elle nécessite une maintenance soutenue de l'âge des faits, cette méthode a par contre l'avantage d'accorder une importance au dynamisme des inférences.

Bien entendu, cette énumération d'heuristiques de résolution des conflits est non exhaustive. Nous invitons le lecteur à trouver une heuristique de résolution des conflits autre que celles ci-haut mentionnées. Ceci lui permettra de mieux réaliser que le choix d'un critère de sélection peut être arbitraire. Les heuristiques que nous avons mentionnées sont le résultat de recherches empiriques, et pour une situation donnée, l'heuristique choisie peut être aussi valide que n'importe quelles heuristiques ci-haut énumérées. Nous verrons à la section 2.2.5 que notre modèle utilise une heuristique hybride.

1.2 Le parallélisme

Ayant établi les différents aspects des antécédents, conséquents, règles, inférences et connaissances, nous allons maintenant centrer notre intérêt sur le parallélisme dans les

moteurs d'inférence en introduisant certaines notions de base. Nous allons définir le parallélisme de données, le parallélisme de traitement, le traitement réparti et le traitement parallèle, l'utilisation d'un espace tuples ou mémoire partagée, la mémoire répartie et la communication par messages.

1.2.1 Le parallélisme de données

Le parallélisme de données comme son nom l'indique est un parallélisme centré sur les données du problème à résoudre. On y morcelle le problème en sous-problèmes ayant un nombre réduit de données à traiter. Les problèmes tout désignés pour ce type de parallélisme sont ceux dont le traitement à effectuer est très localisé à un sous-ensemble de données, c'est-à-dire que le traitement d'une partie des données est indépendant de celui du reste des données. Il est alors aisé de diviser le problème en morceaux et de demander à différents processeurs d'effectuer le traitement sur ceux-ci. Chaque processeur effectue un traitement semblable sur des données différentes.
Le traitement d'images et le calcul de fractals sont des exemples qui se prêtent bien au parallélisme de données, car le calcul d'une région de l'image n'a pas de répercussion sur le calcul d'une autre région.

1.2.2 Le parallélisme de traitement

Le parallélisme de traitement est moins commun. Il s'agit de morceler le problème à résoudre en sous-tâches; chaque processeur doit alors effectuer un traitement qui lui est particulier mais sur l'ensemble des données. La sérialisation des opérations nécessite une gestion plus serrée du travail de chaque processeur. Par exemple, dans le cas d'un moteur d'inférence, un processeur peut être en charge de l'appariement, un autre des entrées/sorties et un autre de l'exécution des règles.

La compilation est un problème où le parallélisme de traitement serait avantageux; un

processeur peut s'occuper de la création de la table des symboles, un autre effectue l'optimisation des boucles et un dernier produit du code objet. Mais, dans tous les cas, le parallélisme de traitement est plus difficile à accomplir que le parallélisme de données notamment dû à la difficulté d'établir correctement le temps que nécessite l'exécution d'un traitement versus le temps d'exécution d'un autre traitement. Immanquablement, un processeur n'aura rien à faire en attente de la fin du traitement d'un autre.

1.2.3 Le traitement réparti

Le traitement réparti comprend souvent la notion de *maître-esclaves.* Un processeur *maître* gère la gestion du problème en distribuant des tâches aux autres processeurs qu'on nomme *esclaves.* Lorsqu'il a terminé son travail, l'esclave rapporte le résultat au maître et se voit affecté à une nouvelle tâche. Le traitement réparti est particulièrement difficile à accomplir principalement dû au problème de variables partagées; l'article de Brogi [Bro91] en discute longuement et présente une forme innovatrice de traitement réparti en éliminant le problème des variables partagées. C'est l'approche qu'utilise *Rational Rose* [Boo94].

1.2.4 Le traitement parallèle

Le traitement parallèle peut être subdivisé en deux catégories différentes, l'une logicielle, l'autre matérielle ; leur point commun étant de travailler à résoudre le même problème.

Traitements distinctifs : Dans l'approche de traitement parallèle à traitements distinctifs, plusieurs processeurs se voient affecter la résolution d'un même problème en utilisant des approches différentes. Le premier qui arrive à un résultat avise les autres de ne plus chercher. Dans cette méthode de résolution de problèmes, l'approche de chaque processeur est distincte.

Traitements spécialisés : L'environnement matériel de cette approche implique une certaine hétérogénéité de processeurs et chaque processeur spécialisé effectue une tâche spécifique pour laquelle il excelle. Un processeur peut être en charge du déchiffrement de messages, un autre expert en calcul à virgule flottante. Un processeur maître distribue les tâches selon les spécialités de chacun.

1.2.5 L'espace tuples

Aussi connu sous le nom de *Blackboard*, l'espace tuples est un espace de mémoire partagée par tous les processeurs. Chaque processeur peut accéder directement (ou via des primitives de communication qui en simulent l'accès direct) à cet espace. Cet espace est utilisé pour partager de l'information sous forme de variables ou de structures. Les structures de tuples sont généralement utilisées. La grande force de l'espace tuples est l'accès aux données; chaque processeur peut en effet accéder à toutes les données du problème. Mais sa grande force est aussi sa faiblesse; par respect de l'intégrité des données, un seul processeur peut effectuer un traitement sur une donnée à chaque instant. Ceci oblige les autres processeurs qui désirent obtenir l'exclusivité de cette information à attendre, réduisant d'autant le gain de performance offert par l'espace tuples. Chaque processeur doit extraire le ou les tuples dont il a besoin, effectuer son traitement sur les données recueillies et déposer les tuples modifiés dans l'espace tuples. Dans certains modèles, la gestion d'un tel espace n'est le travail d'aucun processeur en particulier; dans d'autres, c'est le travail d'un processeur dédié. L'espace tuples comporte plusieurs avantages.

L'exclusion mutuelle : Il est aisé de gérer l'exclusion mutuelle des données. Il suffit de retirer le tuple de l'espace pour être le seul à pouvoir effectuer un traitement.

La connaissance commune : Chaque processeur a accès à toutes les données du problème. Ceci ne contredit pas la possibilité déjà vue d'utiliser l'exclusion mutuelle.

La gestion de l'espace : En général, des mécanismes découplés de l'espace tuples sont fournis et le moteur d'inférence n'a pas à se soucier de sa gestion.

La gestion de l'espace tuples, quoique relativement aisée, n'en demeure pas moins coûteuse [Ale92a]. À noter aussi que, dans certains environnements d'espace tuples (notamment Brenda d'ALEX-Trollius), il est parfois nécessaire de remplacer un tuple par un autre (bidon celui-là) afin d'éviter le blocage. Dans l'environnement Brenda d'ALEX-Trollius, un processeur qui tente d'aller chercher un tuple non présent dans l'espace tuples demeure alors bloqué jusqu'à ce que le tuple devienne disponible (ce qui pourrait à la limite ne jamais se produire). L'introduction d'un tuple bidon permet alors au processeur d'extraire le tuple, le reconnaître comme tuple bidon, le redéposer dans l'espace et continuer le déroulement de son exécution. La charge de travail vouée à la manipulation des tuples est ainsi augmentée. Lorsque le tuple est en grande demande par tous les processeurs, ils vont constamment aller chercher ce tuple bidon et le remettre jusqu'à ce que le vrai tuple soit remis dans l'espace tuple. Évidemment, dans ce cas, les performances se dégradent et l'espace tuples n'est plus la solution à retenir.

1.2.6 La mémoire distribuée

La mémoire distribuée peut se présenter sous plusieurs formes. Dans un cas, chaque processeur dispose de sa mémoire et doit en effectuer la gestion; dans un autre, la mémoire locale de chaque processeur est accessible à tous. Parfois, l'espace tuples est implanté ainsi. C'est notamment le cas de l'espace tuples d'ALEX-Brenda [Ale92a]. L'avantage marqué de cette technique est de permettre une grande flexibilité d'utilisation. Si, pour certains problèmes, on ne désire pas utiliser l'espace tuples, alors chaque processeur dispose de son espace mémoire. Si par contre l'utilisation d'un espace tuples est jugé à propos, alors des mécanismes de communication élaborés permettent à chaque processeur de 'voir' l'espace mémoire de tous comme un seul et unique espace de mémoire partagée.

1.2.7 La communication par messages

La communication par messages est un des grands paradigmes du parallélisme. Il consiste à utiliser des primitives du langage de programmation afin de pouvoir aviser d'autres processeurs d'un événement ou fait. Dans l'architecture d'ALEX-Trollius, la communication par messages peut se faire à plusieurs niveaux. Nous ne considèrerons que les messages de niveau 'réseau'. À ce niveau, les messages sont adressés directement au processeur concerné sans tenir compte de la topologie de connexion des processeurs. Nous présentons ici quelques avantages de la communication par messages :

Trafic réduit : Un processeur peut décider d'aviser seulement certains autres processeurs des nouveaux faits qu'il vient de déduire, limitant ainsi le transfert de l'information au minimum.

La synchronisation : Les messages peuvent être porteurs d'information de synchronisation entre les inférences, afin de maintenir au minimum les incohérences entre les différentes bases de faits. Car les bases de faits distribuées sont constamment en état d'incohérence relative.

L'indépendance des données: Un processeur peut détenir l'exclusivité sur certaines variables et ne jamais les partager. La notion de message est elle-même porteuse d'indépendance. Lorsqu'un processeur a expédié un message, il est alors libre de continuer l'évaluation des faits dont il dispose.

Si la cohérence entre les bases de faits distribuées est capitale, alors un processeur devra procéder à l'inférence en utilisant des algorithmes d'engagement multi-phases classiques [Cor81]. De tels algorithmes peuvent entraîner une dégradation des gains de performance qu'offre le parallélisme.

Chapitre 2

Modèle du moteur d'inférence

Dans ce chapitre, nous présentons notre modèle de moteur d'inférence. Nous commençons par décrire certains aspects syntaxiques et sémantiques du langage accepté par notre moteur d'inférence. Nous présentons ensuite le modèle fonctionnel du moteur, soit l'utilisation des fichiers de paramètres, les conditions d'appariements et le déroulement des inférences. Nous couvrons par la suite les différents types de fonctions, le choix des heuristiques utilisées et la communication inter-noeuds. Pour terminer, nous discutons de la communication, de l'affichage et du contrôle avec VAPS, et nous expliquons la représentation interne utilisée.

2.1 Aspects syntaxiques et sémantiques des fichiers de paramètres

Notre moteur d'inférence accepte, en paramètre, un problème à résoudre. Ce problème est décrit à l'aide de quatre fichiers : un fichier de *noms*, un fichier de *règles*, un fichier de *faits* et un fichier de *routage*. Ce sont ces quatre fichiers de paramètres qui permettent à notre moteur d'être générique. À la section suivante, nous décrivons chaque fichier. Nous présentons ici les notions nécessaires pour bien utiliser les fichiers de paramètres et

la sémantique associée à la syntaxe admise pour chacun de ces fichiers. La syntaxe sous forme Backus-Naur est présentée en annexe 5.

Avant de discuter de la sémantique associée à la syntaxe, nous expliquons en premier lieu la lecture des fichiers de paramètres. La lecture des fichiers de paramètres s'effectue par *ligne* et chaque ligne est constituée d'une suite de *jetons*. Un *jeton* est une suite de caractères alphanumériques ne contenant aucun espace. L'espace est réservé pour délimiter les jetons.

Pour ce qui est des aspects sémantiques associés à la syntaxe des fichiers, nous mettons en évidence les notions suivantes.

- **Les commentaires:** Dans les fichiers de paramètres, les commentaires doivent commencer par la chaîne "// " (l'espace, délimiteur de jeton, est essentiel). Après lecture de ce jeton, le reste de la ligne est ignoré; les commentaires ne sont pas conservés et aucune référence ne peut y être faite.

- **Les types de caractères:** Les mots clés des noms de liste dans le fichier des noms, les noms de processeurs dans le fichier de routage et les mots réservés dans le fichier de règles doivent être en majuscules. Les noms des différentes fonctions prédéfinies doivent être lexicalement identiques à leur appellation originale. Une liste complète de ces différents mots clés est donnée à l'annexe 5. Nous avons opté pour des noms en majuscules pour les *objets*, les *attributs*, les *mutateurs* et les *constantes*. Les *variables* commencent par le caractère $ suivi d'un nom en minuscules. Notre langage différencie les minuscules des majuscules.

- **L'introduction des variables:** Les règles sont lues dans l'ordre normal de lecture d'un fichier. Les variables doivent être introduites dans la section des *antécédents* d'une règle. Toutes les variables doivent apparaître pour la première fois

dans les *tuples* avant de pouvoir être utilisées dans une fonction. Il n'y a qu'une exception à cette règle, nous l'expliquerons plus loin. L'exemple suivant montre l'introduction d'une variable dans un antécédent de règle :

POMPE ETAT *$etat*

Dans ce cas, la variable *$etat* est introduite à la position *valeur* dans le triplet *objet, attribut, valeur.* On peut introduire une variable à la place de n'importe lequel des trois champs d'un antécédent, on peut aussi introduire plus d'une variable par antécédent.

L'exception dont nous avons parlée se présente dans les fonctions, le paramètre de *sortie* (le troisième paramètre) d'une fonction peut introduire une variable, c'est le seul cas d'utilisation d'une variable non introduite dans les antécédents. Prenons l'exemple suivant :

Plus $entrée_1 $entrée_2 $sortie

Les variables *$entrée_1* et *$entrée_2* doivent être définies avant d'être utilisées sinon le résultat obtenu est indéterminé. Le paramètre *$sortie*, qu'il soit instancié ou non, prendra la valeur résultante de l'exécution de la fonction *Plus* de notre exemple.

2.2 Modèle fonctionnel

Le moteur d'inférence que nous avons développé possède toutes les grandes caractéristiques d'un moteur classique que nous avons vu au chapitre 1, ainsi que certains aspects particuliers. La première opération effectuée par chaque moteur sur chaque processeur est la lecture des différents fichiers de paramètres. Vient ensuite le cycle d'évaluation présenté à la figure 3. Le cycle est composé des phases d'appariements, de la création d'un ensemble des conflits et de la résolution des conflits ; le moteur vérifie ensuite s'il

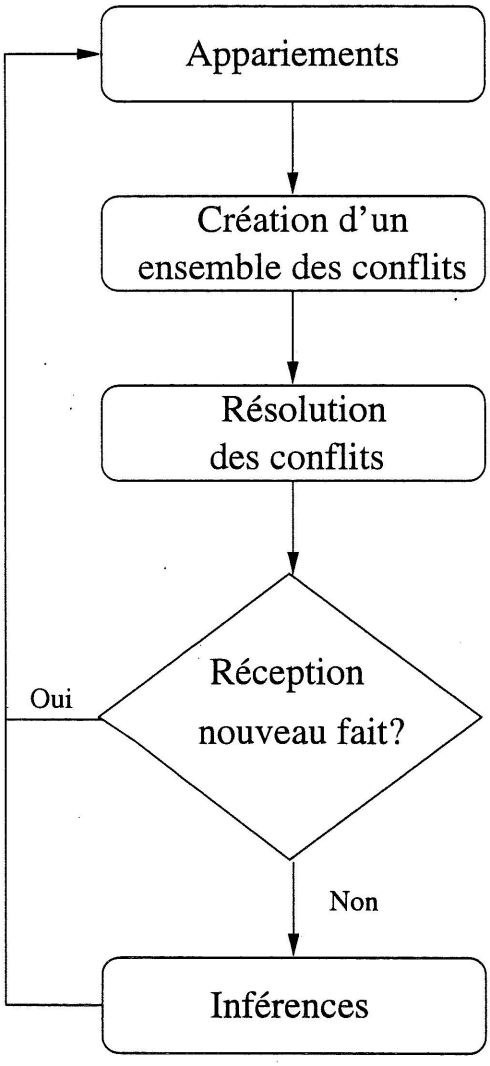


Figure 3: Cycle d'évaluation

possède bien les faits les plus récents, puis il passe à la phase d'inférence ou d'exécution de la règle choisie.

2.2.1 Les fichiers de paramètres

Chaque fichier peut être vide ou contenir des informations. Nous présentons séparément le fichier des *noms*, le fichier des *règles*, le fichier des *faits* et le fichier de *routage*.

- **Le fichier des noms :** Le fichier des *noms* est connu de tous les processeurs. Il contient, comme son nom l'indique, les listes de tous les noms d'objets, attributs, constantes, variables, fonctions et règles, utilisés dans les autres fichiers de paramètres. Le fichier de noms contient aussi les valeurs à associer aux différentes constantes. Il peut aussi contenir des lignes de commentaires.

  Chaque énumération doit commencer par un mot clé indiquant à quelle liste le programme doit associer les différents noms présentés. De plus, chaque liste doit se terminer par le mot clé FIN_TABLEAU. Les sept listes : OBJETS, ATTRIBUTS, FONCTIONS, VARIABLES, CONSTANTES, VAL_CONSTANTES et REGLES doivent être présentes dans ce fichier. S'il n'y aucun nom pour une liste donnée, le mot-clé de la liste suivi du mot FIN_TABLEAU doivent néanmoins y être présents. L'ordre de présentation des listes n'a pas d'importance et, pour la quasi totalité des listes, l'ordre des noms n'a pas d'importance non plus. La seule exception à cette règle concerne la liste des valeurs des constantes : cette liste doit être une suite d'entiers dont la position des valeurs correspond à la position des noms des constantes dans la liste CONSTANTES. L'exemple suivant illustre cette contrainte :

  CONSTANTES ZERO CENT FIN_TABLEAU

  VAL_CONSTANTES 0 100 FIN_TABLEAU

  Dans cet exemple, la constante ZERO prendra la valeur 0 pour toutes ses utilisations par les fonctions des règles du problème. Il va de soi que CENT aura la valeur 100.

Si une constante nommée n'a pas de valeur spécifiée, le résultat de l'exécution d'une fonction utilisant une telle constante est indéfini ; aucune valeur par défaut n'est associée à une constante.

- **Le fichier des règles :** Comme tous les autres fichiers de paramètres, le fichier des règles peut être vide. Il va de soi qu'un fichier de règles vide destiné à un processeur exécutant un moteur d'inférence à base de règles est inutile.
  Une règle est définie par son nom suivi de sa priorité statique ; vient ensuite le mot-clé IF suivi d'un ou plusieurs antécédents. Le mot-clé THEN signale la fin de la liste des antécédents et le début de l'énumération des conséquents. Finalement, le mot-clé FI indique la fin de la règle.

Nous obtenons donc la structure générale suivante :

```
// Une description textuelle de la règle
nom de règle    priorité statique
IF
{antécédent}*
THEN
{conséquent}*
FI
```

Figure 4: Structure d'un fichier de règles

Des commentaires peuvent être intercalés à n'importe quel endroit et non seulement entre les règles. La section des heuristiques 2.2.5 explique l'utilisation des priorités statiques.

- **Le fichier de faits :** Ce fichier contient les faits initiaux. C'est à l'aide de ces faits que le moteur effectue ses premières inférences. La présence d'un fichier de faits vide veut dire qu'il est possible que les connaissances requises à l'inférence de nouveaux faits, par un processeur donné, ne soient pas disponibles au départ. Ces connaissances lui sont fournies par des messages venant des autres processeurs au cours de leur exécution. Les faits décrits dans ce fichier ne peuvent pas contenir de variables, mais ils peuvent contenir des constantes telles que définies dans le fichier des noms. Dans ce fichier comme dans les autres, la présence de commentaires est admise et ces derniers ne sont pas conservés. La figure 5 présente un exemple de fichier de faits :

```
// Le débit de la pompe correspond à la
// capacité de pompage en litres par minute.
POMPE DEBIT 24
POMPE ETAT ON
EAU NIVEAU 50
```

Figure 5: Exemple d'un fichier de faits

- **Le fichier de routage :** Le fichier de routage contient les informations de routage des inférences. Ce fichier est constitué d'une suite de triplets *objet, attribut, processeur.* Ainsi, si le résultat d'une inférence provoque la modification de la valeur du champ *attribut* d'un *objet*, celle-ci sera communiquée au(x) *processeur(s).* Si un fait doit être communiqué à plusieurs processeurs, il doit y avoir plusieurs entrées ayant le même *objet* et *attribut.* Si un fait concerne le processeur qui effectue l'inférence, un triplet *objet, attribut, processeur* doit néanmoins l'indiquer explicitement. L'association *objet, attribut, processeur* permet de limiter le nombre de

messages à transmettre, car un processeur donné ne reçoit que les faits dont il a besoin, c'est-à-dire uniquement ceux dont un appariement pourrait être possible compte tenu de sa base de règles. Nous présentons ici un extrait d'un fichier de routage :

```
// Le processeur P1 gère l'état et le débit
// de la pompe.
POMPE DEBIT P1
POMPE ETAT P1
// Aviser P1,P2 et P3 de tout changement du niveau
// d'eau.
EAU NIVEAU P1
EAU NIVEAU P2
EAU NIVEAU P3
```

Figure 6: Exemple d'un fichier de routage

Dans cet exemple, trois processeurs ont besoin de connaître le niveau d'eau et un seul a besoin de connaître l'état et le débit d'une pompe.

2.2.2 Appariements

L'appariement étant une opération critique de notre moteur, nous définissons ici la *table d'appariements* ainsi que les différents types d'appariements possibles. La table d'appariement constitue le coeur de notre moteur d'inférence. Nous devons y accorder toute l'attention requise. Ainsi, nous utilisons des segments de pseudo-code afin de décrire les différents traitements qui y sont rattachés.

Table d'appariements

La table d'appariements contient toutes les informations requises pour permettre au moteur d'inférence de fonctionner. Cette table contient tous les indices des différents antécédents, conséquents, faits et règles du problème. Elle est constamment en changement de taille et de contenu selon les faits du moment.

Tableau 1: Table d'appariements

tup	**fct**	**rul**	**sta**	**dyn**	**tag**	**seq**	**flg**
0	0	2	2	0	0	0	0
1	1	2	2	0	0	0	0
3	6	2	2	0	0	0	0
6	5	3	2	0	0	0	0
11	1	9	0	0	0	0	0
13	2	9	0	0	0	0	0

Considérant qu'une règle est représentée par une série d'antécédents et de conséquents, et considérant qu'un fait peut s'apparier à zéro, un ou plusieurs antécédents d'une règle nous avons convenu d'utiliser une table pour représenter ces relations d'antécédents avec faits. Le tableau 1 représente une petite section d'une table d'appariements, les valeurs présentées ne sont pas toutes réelles car, à l'interne, certains champs sont doublés d'une valeur de contrôle; la section 2.4 sur la représentation interne traite de ces valeurs de contrôle. Chaque ligne du tableau représente une relation entre un antécédent et un fait.

Chaque ligne contient les informations sur la règle à laquelle appartient le tuple mentionné à la colonne **tup**. Chaque ligne contient aussi l'indice du fait dans la table des faits et trois champs de contrôle requis pour le bon déroulement des appariements. Pour chaque ligne, le tuple **tup** peut être apparié individuellement au fait **fct**. Le champ **tup** indique l'indice du tuple dans la table des tuples. Le champ **fct** est l'indice du fait (dans

la table de faits) auquel le tuple **tup** peut être apparié. Le chiffre que l'on retrouve dans le champ **rul** représente l'indice de la règle dans la liste REGLES du fichier des noms. Le champ **sta** représente la priorité statique de la règle telle que définie dans le fichier des règles. Le champ **dyn** indique la priorité dynamique de la règle.

Le champ **tag** est utilisé pour marquer une règle comme ayant tout ses tuples satisfaits. Le champ **seq** est utilisé pour indiquer l'ordre des appariements inter-éléments. Ce champ **seq** sert aussi lors des retours arrière lorsque le moteur procède à l'élaboration de l'ensemble des conflits. Finalement, le champ **flg** indique qu'une règle fait partie de l'ensemble des conflits. À chaque cycle d'évaluation, les champs **tag, seq** et **flg** sont remis à zéro. Si une règle est choisie, sa priorité dynamique est mise à zéro alors que la priorité dynamique de toutes les autres règles faisant partie de l'ensemble des conflits est incrémentée de un. C'est ce procédé de vieillissement qui assure que, si les conditions d'activation de plusieurs règles de même priorité statique sont toujours présentes, chacune de ces règles sera éventuellement choisie pour l'exécution.

Notez que la table d'appariements n'est pas une table normalisée et que cette table contient une entrée par appariement différent. Ainsi, si un tuple est appariable à trois faits, la table contiendra trois entrées pour ce tuple.

Types d'appariements

Le moteur d'inférence doit procéder à une phase d'appariements entre sa base de faits et l'ensemble des règles dont il a la gestion pour déterminer s'il y a lieu de procéder au calcul de l'ensemble des conflits. Quatre cas d'appariement peuvent se produire : l'*appariement de deux constantes*, l'*appariement d'une variable et d'une constante*, l'*appariement d'une variable à un objet, attribut ou une valeur* et finalement l'*appariement d'une valeur avec une constante*. Dans cette section, nous présentons la table d'appariements et nous

élaborons sur chacun des quatre types d'appariements.

- **Appariement de deux constantes :** Deux constantes sont compatibles si et seulement si elles sont identiques. Il faut comprendre ici que les *valeurs* associées aux constantes ne sont pas vérifiées. Pour être appariables, les constantes doivent être lexicalement identiques dans les différents fichiers de paramètres. L'exemple suivant illustre un appariement réussi des deux constantes ON :

$$\overbrace{\text{POMPE \$etat ON}}^{antécédent} \text{ et } \overbrace{\text{POMPE ETAT ON}}^{fait}$$

  Les constantes ne peuvent se trouver que dans le champ *valeur* d'un fait ou tuple comme dans l'exemple précédent ou comme paramètre d'une fonction comme dans l'exemple qui suit :

$$\overbrace{\text{Plus(\$Niveau,DELTA,\$NewNiveau)}}^{fonction}$$

  Ici la constante DELTA devrait être définie dans la liste des constantes.

- **Appariement d'une variable et d'une constante :** Lorsqu'une variable s'apparie à une constante, elle prend la *valeur* de la constante. Le moteur récupère la valeur associée à la constante, telle que définie dans le *fichier des noms.* Si un nouveau fait est inféré à l'aide de cette information, ce fait contient une *valeur* et non pas une constante. Dans l'exemple illustré aux figures 7 et 8, une variable, DEBIT, est instanciée à l'aide de la valeur 5 de la constante CINQ.

- **Appariement d'une variable à un objet, un attribut ou une valeur :** Dans le cas où la variable s'apparie à un objet, un attribut ou une valeur, la variable devient identique à son appariement. Dans l'exemple suivant, la variable \$obj s'identifie à l'objet METHANE et la variable \$att s'identifie à l'attribut ALARME.

$$\overbrace{\text{\$obj \$att ON}}^{antécédent} \text{ et } \overbrace{\text{METHANE ALARME ON}}^{fait}$$

```
CONSTANTES UN DEUX CINQ NIVEAU_DE_DEPART FIN_TABLEAU
VAL_CONSTANTES 1 2 5 10 FIN_TABLEAU
```

Figure 7: Définition de quelques constantes

```
// Fichier des faits initiaux
...
POMPE DEBIT CINQ
...
```

Figure 8: Sous-ensemble d'un fichier de faits

On ne peut pas associer une variable à un *mutateur* ou à une *fonction.*

- **Appariement d'une valeur et d'une constante:** Il y a appariement d'une valeur dans un antécédent à une constante, seulement si la valeur associée à la constante est identique à la valeur de l'antécédent. Pour l'exemple donné ci-dessous, il y a appariement entre :

$$\overbrace{\text{EAU NIVEAU NIVEAU\_DE\_DEPART}}^{antécédent} \; et \; \overbrace{\text{EAU NIVEAU 10}}^{fait}$$

si et seulement si il existe une concordance dans le fichier des noms permettant d'associer le niveau de départ et 10 comme dans l'extrait du fichier des noms présenté à la figure 7.

2.2.3 Inférences

Les inférences permettent de modifier la base des faits. Chaque moteur effectue ses inférences séquentiellement et tous les processeurs travaillent en parallèle. Il n'y a pas d'ordonnancement des différentes inférences effectuées sur les différents processeurs. Un tel ordonnancement, si souhaitable, relève du rôle du programmeur des règles : voir la section 2.2.5 sur les heuristiques. Nous discuterons ici du déroulement du processus d'inférence sur un processeur, de l'ordre d'évaluation des antécédents, des fonctions, du calcul de l'ensemble des conflits, de la résolution des conflits et finalement de l'évaluation des conséquents.

Les moteurs d'inférence qui s'exécutent sur tous les processeurs sont identiques. Tous possèdent la même boucle d'évaluation (voir figure 3). En premier lieu, un moteur lit ses fichiers de paramètres du problème, comme nous l'avons vu à la section 2.1, certains de ces fichiers sont les mêmes pour tous les processeurs et certains sont propres à chacun. Après l'étape de lecture, le moteur possède une base de règles, une base de faits et une table de routage ainsi que l'ensemble des mots utilisés dans le problème. Il entre alors dans la boucle d'évaluation schématisée à la figure 3. À la phase d'appariements, seuls les nouveaux faits sont traités (au démarrage du moteur, tous les faits sont nouveaux). Le moteur procède ensuite à la mise à jour de sa table d'appariements. La table d'appariements contient toutes les informations sur les relations à établir entre les faits, les tuples des règles, les priorités statiques et dynamiques ainsi que divers autres champs de contrôle nécessaires à la gestion adéquate des inférences. La table d'appariements est construite à partir de la base de faits et de la base de règles. Par conséquent, elle est donc en constante mutation; dès que la base de faits est modifiée, la table d'appariements est mise à jour.

Modification de la base de faits

Suite à la modification d'un fait, le moteur compare celui-ci à tous les antécédents de sa base de règles. S'il y a appariement entre le fait et un tuple, ce tuple est marqué comme *potentiellement satisfait.* La notion *potentiellement satisfait* signifie simplement que le fait peut instancier le tuple. Cela ne signifie pas que la règle à laquelle appartient ce tuple est instanciable, car il est possible que l'instanciation ultérieure de ce tuple par ce fait entre en conflit avec l'instanciation d'autres tuples d'une même règle.
L'opération de comparaison est répétée jusqu'à épuisement des tuples et des faits modifiés.

L'évaluation des antécédents :

L'évaluation des antécédents et conséquents de règles nécessite l'utilisation d'une table nommée *table des variables.* Avant de procéder à la présentation du mode d'évaluation des antécédents, nous présentons cette table.

Table des variables : Le tableau 2 représente une partie de la table des variables lors de l'évaluation d'antécédents d'une règle. Cette représentation abstraite de la table figurative nous montre trois colonnes, soit les colonnes **variables**, **liaison** et **temporaire.**

Tableau 2: Table des variables

variables	liaison	temporaire
$state	1	-
$input	9	-
$level	73	-
$newlevel	-	82

La première colonne contient le nom de chacune des variables du problème. Le contenu de cette colonne est identique à la liste des variables que l'on retrouve dans le fichier des

noms. La seconde colonne, que nous avons nommé *liaison*, contient les valeurs de variables définitives (pour une règle donnée et à un moment donné) que l'on peut utiliser pour instancier une règle. La troisième colonne, la colonne *temporaire*, représente les valeurs de variables provisoires. Nous procédons comme suit. Premièrement, lorsqu'un antécédent de règle peut s'apparier à un fait et que cet antécédent contient une ou plusieurs variables, alors nous plaçons la valeur associée à chacune des variables dans la colonne *temporaire*, à chacune des lignes appropriées. Nous effectuons un premier mouvement de la colonne temporaire vers la colonne liaison et réinitialisons la colonne temporaire. Ce mouvement s'effectue sans restriction, car la colonne liaison ne contient rien à cette étape. Nous procédons à l'appariement des antécédents successifs de façon similaire, c'est-à-dire nous copions la valeur de chaque variable appariable à un fait à la ligne de cette variable, dans la colonne temporaire. Mais cette fois avant d'effectuer le mouvement de la colonne temporaire à la colonne liaison, nous nous assurons qu'il n'y a pas de conflit (deux valeurs non nulles différentes) entre les deux colonnes de valeurs pour chacune des variables. S'il n'y a pas de conflit entre les deux colonnes alors nous effectuons le mouvement (copie de la colonne temporaire dans la colonne liaison et réinitialisation de la colonne temporaire). Parfois ce mouvement remplace une valeur par une autre identique[1], parfois il initialise une nouvelle cellule de la colonne liaison avec une valeur. Si par contre, il y a conflit entre les colonnes temporaire et liaison alors le fait courant est rejetté (du moins temporairement) et l'algorithme procède à la recherche d'un autre fait pouvant s'apparier à l'antécédent courant. Lorsqu'un fait est trouvé, les algorithmes d'association (copie de la valeur à la cellule temporaire appropriée) et de validation (vérification de conflit) sont répétés jusqu'à épuisement des antécédents d'une règle.

[1]Nous considérons que dans ce contexte il est moins coûteux d'effectuer la substitution systématique que de comparer continuellement et ne substituer qu'au besoin.

Considérons le cas limite suivant. L'épuisement des faits s'effectue avant l'appariement réussi d'un antécédent d'une règle. L'algorithme procède alors à une reprise arrière incrémentale. Il rejette le dernier fait qu'il avait accepté et tente un nouvel appariement avec un nouveau fait, si cela ne fonctionne pas, il recule récursivement en rejettant cette fois les deux derniers faits qu'il avait acceptés et procède ainsi de suite jusqu'à épuisement récursif des faits ou jusqu'à un appariement réussi. Rejeter un fait accepté, signifie réinitialiser la ou les variables de la colonne liaison que ce fait avait réussi à faire adopter. Si après avoir terminé l'algorithme d'appariement, un antécédent demeure sans appariement réussi, alors la règle est ignorée pour ce cycle d'évaluation. Si par contre, tous les antécédents d'une règle ont trouvé un fait à qui ils peuvent s'apparier sans conflits alors la règle est marquée instanciable. Seules les règles instanciables (donc sans conflits internes) font partie de l'*ensemble des conflits*.

L'évaluation des fonctions

L'évaluation des fonctions présentes parmi les antécédents est très simple. Les fonctions sont présentes dans la table d'appariements et aucun fait ne leur est associé. La procédure de la figure 9 indique comment s'effectue l'évaluation d'une fonction.

Calcul de l'ensemble des conflits

Chaque règle, ayant au moins un fait pouvant instancier chacun de ses antécédents, est évaluée pour déterminer si elle est instanciable. Lorsqu'une règle est instanciable, un numéro d'instanciation lui est affecté. Toutes les règles possédant un numéro d'instanciation forment l'ensemble des conflits.

La résolution des conflits

Le choix de la règle à exécuter parmi celles présentes dans l'ensemble des conflits est basé sur trois heuristiques décrites à la section 2.2.5. Il suffit de mentionner ici qu'à la suite

La fonction a-t-elle des paramètres de type 'variable'?

Si oui

Récupérer les valeurs associées à ces variables dans la table temporaire des variables

La fonction a-t-elle des paramètres de type 'constante'?

Si oui

Récupérer les valeurs associées à ces constantes telles que définies dans le fichier des noms.

Trouver la fonction devant être appelée dans le tableau des fonctions prédéfinies.

Appeler la fonction avec les valeurs des paramètres récupérées.

La fonction retourne-t-elle VRAI?

Si oui

La fonction possède-t-elle un paramètre de retour?

Si oui

Le copier dans la table des variables.

Poursuivre l'évaluation

Si non

Marquer cet antécédent non satisfait

Tenter un retour arrière (chercher à associer d'autres faits/antécédents afin que les paramètres de la fonction changent et permettent ainsi une réévaluation.)

Figure 9: Procédure d'évaluation d'une fonction

de la résolution des conflits, une et une seule règle sera choisie pour être exécutée.

L'évaluation des conséquents

L'évaluation des conséquents est vue comme une opération atomique, localement, et comme une suite d'opérations sur les sites éloignés. Le processeur local évalue tous les conséquents avant de procéder à un nouveau calcul de son ensemble des conflits. Par contre, pour les processeurs éloignés, l'utilisation du paradigme de la communication par messages leur présente les nouveaux faits comme une suite d'éléments de connaissances détachés. Théoriquement, un processeur éloigné pourrait avoir le temps d'effectuer un cycle d'évaluation et d'inférence entre la réception d'un premier fait d'une règle et la réception d'un autre fait le concernant et ayant été inféré par la même règle. C'est un cas de demie-vérité. Prenons l'exemple d'un courtier qui reçoit une note disant "Vendez toutes mes actions" et qu'après qu'il ait procédé, il reçoit le reste du message "dès qu'elles vaudront $20.00 pièce". Des recherches futures sur ce sujet permettraient d'évaluer l'impact de ce problème et d'y trouver une solution. Nous élaborons plus en détails sur cette question à la conclusion du mémoire.

Généralement, les conséquents sont composés de *mutateurs* et de faits. Les mutateurs permettent de modifier la base de faits, qu'elle soit locale ou éloignée. Une modification de la base de faits n'aura lieu que si le nouveau fait constitue une *modification* de la valeur de l'attribut par rapport au fait présentement dans la base. Si le fait reçu est identique à celui présent dans la base, alors il n'y a pas de recalcul de la table d'appariements ni de l'ensemble des conflits.

Condition d'arrêt

Dans l'introduction, nous avons vu qu'un moteur d'inférence type répète son cycle d'évaluation jusqu'à ce qu'une condition d'arrêt soit satisfaite. Cette condition d'arrêt est générale-

ment l'atteinte d'un but donné ou l'épuisement de toutes les inférences possibles. Notre modèle de moteur d'inférence a la particularité de ne pas s'arrêter, même s'il n'y a plus d'inférence possible. À l'épuisement des inférences, notre moteur reste en attente de nouveaux faits provenant des autres processeurs, car ceux-ci peuvent toujours venir alimenter un moteur en famine de nouveaux faits par leurs diverses inférences.

2.2.4 Les fonctions

Un certain nombre de fonctions mathématiques et booléennes sont prédéfinies. La définition de ces fonctions est contenue dans le fichier *fonctions_predefinies.c.*

Les fonctions de calcul mathématique : Toutes les fonctions mathématiques possèdent deux paramètres en entrée et un paramètre en sortie. Les fonctions mathématiques telles que définies ont toutes une valeur de retour VRAI.

1. **Plus :** Les deux premiers paramètres sont additionnés et le résultat de l'addition est déposé dans le troisième paramètre.
2. **Minus :** Cette fonction soustrait le deuxième paramètre du premier et le résultat est déposé dans le troisième.
3. **Mult :** Effectue la multiplication de deux nombres entiers.
4. **Min et Max :** Ces fonctions déposent, respectivement, dans le paramètre de retour, l'élément minimum et l'élément maximum des deux paramètres d'entrée.

Les fonctions booléennes : Les trois fonctions booléennes prédéfinies *Equal, Greater,* et *Smaller* comparent les deux paramètres d'entrée des fonctions et retournent le résultat de la comparaison des paramètres. Le contenu du paramètre de sortie est indéfini, car il n'est pas utilisé, c'est la fonction elle-même qui retourne le résultat.

Si les fonctions prédéfinies ne suffisent pas pour le besoin d'un problème donné, l'utilisateur doit définir de nouvelles fonctions et les inclure dans le fichier *fonction_predefinies.c.* Il devra évidemment recompiler le programme au complet. L'ordre d'apparition des noms de fonctions dans le tableau des *Fonctions_predefinies* doit correspondre exactement à celui de ces mêmes fonctions dans le *fichier des noms* décrivant le problème à résoudre.

2.2.5 Les heuristiques

Notre moteur d'inférence utilise deux heuristiques pour déterminer quelle sera la règle élue pour exécution parmi les règles présentes dans l'ensemble des conflits. Deux autres heuristiques sont utilisées pour déterminer quelles sont les règles présentes dans l'ensemble des conflits.

- **Les priorités statiques :** Chaque règle possède une priorité statique qui lui a été affectée par le programmeur des règles. Si, dans l'ensemble des conflits, une règle possède une priorité statique supérieure à toutes les autres, cette règle sera choisie.

- **Les priorités dynamiques :** À chaque cycle, la priorité dynamique de chaque règle faisant partie de l'ensemble des conflits et dont l'exécution n'est pas effectuée est incrémentée. Si plusieurs règles ont une priorité statique identique, la règle ayant la priorité dynamique la plus élevée est élue pour exécution. Si plusieurs règles ont, à la fois, les mêmes priorités statiques et les mêmes priorités dynamiques, la première règle rencontrée est choisie pour exécution.

- **Les faits les plus récents :** L'utilisation des faits les plus récents n'est pas considérée dans la résolution des conflits, mais son impact sur la formation du dit ensemble est tel qu'il supplante toutes les autres heuristiques. L'arrivée d'un nouveau fait force un nouveau calcul de l'ensemble des conflits. Cette méthode permet d'assurer une certaine cohérence entre les bases de faits réparties sur chaque processeur. Par contre, cette heuristique risque de provoquer une suralimentation en faits ou une

famine d'exécution. Ce phénomène se produit lorsqu'un processeur n'a que peu de faits et peu de règles à traiter. Sa boucle d'évaluation étant courte, il inonde rapidement les autres processeurs avec ses nouveaux faits provoquant ainsi la réévaluation de leur table d'appariements respective. Il s'en suit une dégénérescence du modèle. Pour pallier à ce problème, nous utilisons un *paramètre de désynchronisation.*

- **Le paramètre de désynchronisation** Le paramètre de désynchronisation est un entier passé à chaque moteur pour les ralentir. Contrairement à l'optique de gain de performance par l'utilisation d'entiers au niveau de la représentation interne, chaque moteur doit être ralenti artificiellement pour deux raisons :

  - **1-** Pour éviter d'inonder de nouveaux faits le processeur (P100) responsable de la communication avec le logiciel d'interface graphique. Si le paramètre de désynchronisation est trop petit, les messages arrivent à une cadence supérieure à la capacité de consommation de messages par l'interface graphique. Il y a débordement de messages et perte de synchronisme relatif entre les bases de faits des processeurs et l'affichage graphique de celles-ci.
  - **2-** Pour ralentir certains processeurs par rapport à d'autres. Ce faisant, on peut s'assurer que tous les processeurs pourront à un moment ou à un autre avoir la chance d'exécuter une de leurs règles. Car à un certain moment un processeur sera prêt à inférer et celui qui avait l'habitude de l'inonder de faits est en boucle d'attente.

2.2.6 La communication inter-noeuds

ALEX-Trollius dispose d'un langage spécialisé appelé *BNAIL*[2] permettant la connexion logique des différents processeurs selon différentes architectures. De plus, plusieurs primitives d'ALEX-Trollius permettent d'établir une communication inter-noeuds à toute une

[2]BNAIL est un acronyme pour *Boot schema Node And Interconnect Language [Ale92a]*

gamme de niveaux. Pour notre moteur, nous avons connecté les différents processeurs avec un schéma *grille (grid)*; les primitives de communication que nous utilisons (voir section 2.3.2) rendent inutile l'utilisation d'une autre architecture de communication.
La communication inter-noeuds est gérée à l'aide du *fichier de routage*. Nous décrivons maintenant les particularités du *fichier de routage*, nous introduisons le principe de *communication par messages* sous ALEX-Trollius, nous discutons des avantages de la relation *objet-attribut-noeud* et finalement nous précisons les actions prises lors d'une *inférence locale*.

Le fichier de routage : Le *fichier de routage* fait partie intégrante de la stratégie utilisée lors d'inférences parallèles. Le *fichier de routage* est particulier à chacun des processeurs ; il contient toutes les informations permettant à un moteur donné de déterminer quels autres processeurs il doit aviser lors d'inférence de nouveaux faits, soit une série de triplets *objet, attribut, processeur*. Chaque processeur ayant son fichier de routage particulier, le programmeur peut donc indiquer l'ensemble minimal de processeurs qui doivent être avisés lors d'une modification. La syntaxe formelle d'un fichier de routage est donnée en annexe 5.

La relation objet-attribut-processeur : Nous avons vu qu'un fait est formé d'un triplet *objet-attribut-valeur*. La relation *objet, attribut, processeur* permet de limiter le nombre de messages inter-noeuds au minimum ; en effet, un processeur donné ne reçoit que les faits pour lesquels au moins une de ses règles de production utilise l'association *objet, attribut*. Cette restriction permet de réduire encore plus le nombre de messages que si on se limitait au niveau *objet*.

L'inférence locale : Lorsque le nouveau fait inféré est utilisé localement, un traitement spécial est effectué par le moteur. Le nouveau fait est traité directement ; un moteur n'envoie pas de messages à lui-même. Chaque nouveau fait inféré localement modifie directement la base de faits et tous les faits produits par une même règle

sont traités avant que le cycle d'inférence ne recommence.

Nous élaborons plus en détails sur la communications inter-noeuds à la section 2.3.2 et sur le format des messages qu'utilise ALEX à la section 2.3.3.

2.3 Communication avec VAPS

La communication entre ALEX et VAPS est établie à l'aide d'un processeur situé sur une carte à l'intérieur d'un SUN ; ce processeur est appelé le *noeud racine*. Le noeud racine communique avec VAPS via les ports de communication TCP/IP. Dans cette section, nous élaborons sur ces aspects ainsi que sur le programme de communication inter-processus, le format de message utilisé et la relation *objet, attribut, membre* reliant les faits aux objets de l'interface VAPS.

- **Le noeud racine :** Le noeud racine est le processeur sur lequel s'exécute le programme effectuant le lien entre ALEX et VAPS. Ce noeud est connu sous le nom de T24, N24 et aussi P100, selon l'utilisation. Il s'appelle T24 dans le fichier *bnail* utilisé au démarrage du système d'exploitation ALEX-Trollius ainsi que dans les fichiers de configuration des connexions (voir les manuels de références d'ALEX-Trollius [Ale92a] pour plus de détails). Lorsqu'on utilise les outils XTrollius, le noeud racine a pour nom N24. Finalement, à l'intérieur des programmes 'c', on utilise la commande *getorigin()* pour obtenir l'adresse du noeud racine. Cette fonction retourne 100, c'est pourquoi les fichiers de routage que nous utilisons mentionnent le noeud racine comme étant P100.

- **Le communication via TCP/IP :** Pour établir la communication entre deux processus s'exécutant sur des ordinateurs SUN, ces derniers doivent utiliser des ports TCP/IP et être connectés logiquement à l'aide d'une table de connexion. La figure 10 présente la partie du fichier */etc/services* ayant trait à la définition

```
# ------------------------------
...
# pour vaps (et vbe)
#
vaps_re 5006/tcp #Port for one VAPS process
vaps_sim 5007/tcp #Port for another VAPS process
...
```

Figure 10: Fichier /etc/services

des ports TCP/IP requis pour établir cette communication. La figure 11 indique le contenu du fichier *vpnetwork*, lequel permet d'établir le lien entre les noms des processus, les machines où s'exécutent les programmes et les ports TCP/IP. Pour cet exemple, *re* est le nom du Runtime Environment de VAPS, *grandduc* le nom de la machine où s'exécute VAPS et *vaps_re* est le nom du port TCP/IP associé. Le programme *root* est le nom donné au programme de lien entre ALEX et VAPS, *nyctale* le nom de la machine ayant la carte ALEX (donc le noeud racine T24) et *vaps_sim* est le nom du port.

- **Les canaux et les membres:** Le mécanisme de communication de VAPS est constitué de canaux et de membres. Un canal défini une collection de données élémentaires échangée comme une seule unité. On nomme *membre* chaque élément de donnée d'un canal. Chaque membre contient une donnée de type élémentaire (c'est-à-dire double, réel, entier court, un long ou un charactère). En plus, chaque canal possède un type et une portée. Un canal peut être de type FAST ou QUEUED. La portée peut prendre la valeur SESSION ou LOCAL. Le programmeur qui utilise un canal de type FAST et de portée LOCAL dispose de fonctions spécifiques pour

```
# ------------------------------
...
# Host Specification Lines
#
# The format of the following lines is as follows:
#
# process_name nick_name host_name port_name
#

 re 1 grandduc vaps_re
 root 2 nyctale vaps_sim

...
```

Figure 11: Fichier /vpnetwork

manipuler les canaux, tandis que la complexité de la gestion des canaux de type QUEUED et SESSION est laissé au programmeur. Les types QUEUED sont utilisés pour établir une communication synchrone. Les canaux FAST, on le devine, sont plus rapides que les canaux QUEUED, mais ces derniers sont plus fiables. La portée LOCAL d'un canal est limitée à une interface tandis que la portée SESSION permet au canal d'être connu de tous les programmes et même au travers d'un réseau. Les membres sont considérés comme des tableaux de données élémentaires. Les tableaux admis par VAPS peuvent être de type vectoriel ou à deux et même trois dimensions. Les interfaces créées par Palmier [Pal95] utilisent des canaux de type FAST, de portée SESSION et ont des membres vectoriels. Pour en connaître plus sur les différentes fonctionnalitées reliées aux canaux et membres le lecteur peut consulter la section 'Network Communication and Applications Interface' de VAPS [Vir93].

- **Le programme de communication inter-processus :** Le programme de communication inter-processus consiste en une boucle d'interrogation entre ALEX et VAPS. Le programme vérifie s'il a reçu un message en provenance de la machine ALEX. Si c'est le cas, il compare ce nouveau fait avec le contenu de sa base de faits ; s'il ne détecte pas de différence ou si ce nouveau message n'a pas d'impact graphique, alors le programme n'effectue aucun traitement avec ce message. Par contre, si le message a un impact graphique, le moteur obtient de sa table de routage le membre sur lequel il doit expédier l'information à l'interface VAPS, convertit le message et l'expédie. Évidemment, pour un message qui modifie un fait, le programme effectue une mise à jour de sa base de faits.

  Similairement, lorsque le programme n'a pas reçu de message d'ALEX, il vérifie s'il a reçu un message de VAPS. Le traitement des messages provenant de VAPS est l'inverse de celui effectué pour les messages d'ALEX : réception de message, conversion en entier, vérification du contenu de la base de faits locale et, s'il y a

lieu, mise à jour de la base de faits, extraction des destinations ALEX et envoi des messages.

Les relations de dépendance du programme de communication inter-processus sont illustrées à la figure 12.

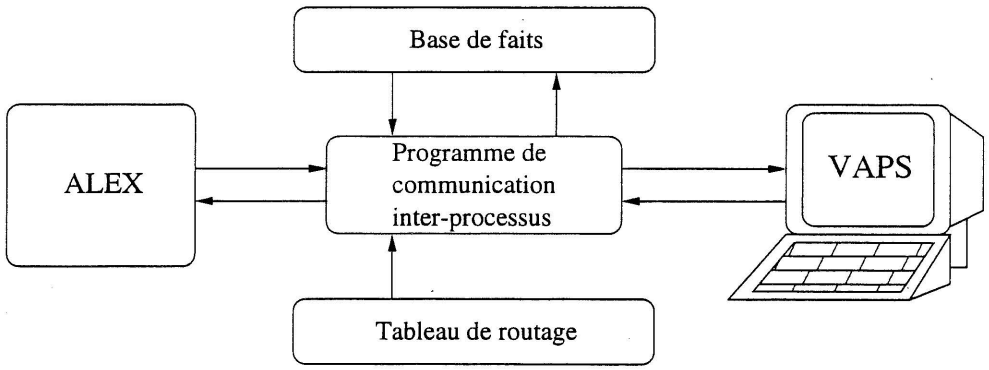


Figure 12: Communication inter-processus

- **La relation objet, attribut, membre:** Chaque objet graphique de l'interface VAPS est relié à un *membre* d'un *canal*. La communication entre les deux applications se matérialise via ces *canaux*. Les canaux de VAPS contiennent des réels et notre modèle n'utilise que des entiers. Aussi est-il nécessaire d'effectuer une conversion entre la lecture et l'écriture ; c'est la partie ALEX qui effectue cette conversion. Une valeur est déposée dans un membre par le processus de communication ALEX-VAPS, l'interface graphique s'en saisit et affecte à l'objet relié le changement d'état demandé. Inversement, l'interface graphique transmet au processeur P100 toute modification demandée par l'utilisateur via un ensemble de membres et canaux de sortie.

- **Le format des messages:** Le format des messages échangés entre le processeur P100 et l'interface graphique est très simple, le programme de communication envoie une structure de message (un canal composé de membres) dans laquelle il

indique quelle valeur réelle il désire affecter à quel membre. L'interface VAPS interpréte le contenu de chaque membre et affecte la valeur à l'objet graphique associé au membre. Pour la réception, le processeur P100 juxtapose le nom d'un l'objet et de son attribut et obtient ainsi le nom du membre associé, il ne lui reste qu'à lire le membre, convertir le réel reçu en entier et affecter la valeur de l'attribut.

2.3.1 Affichage et contrôle

La création d'une interface graphique pour interagir avec le moteur d'inférence parallèle est une tâche qui doit être faite après consultation avec la personne qui programme les règles d'inférence. Leur complicité doit être complète, car les noms donnés aux différents composantes au moment de la conception doivent être rigoureusement respectés dans les deux systèmes.

La conception d'une interface : Le programme qui établit la communication entre VAPS et ALEX permet la conversion des notions distinctes d'*objet* et d'*attribut* que reconnaît ALEX en des concepts plus restrictifs d'association *objet-attribut* que reconnaît VAPS. Même si conceptuellement pour le moteur d'inférence, un *objet* et son *attribut* sont deux entités distinctes (on peut avoir un objet avec plusieurs attributs), pour VAPS la relation *objet, attribut* constitue une et une seule entité. Par exemple, la notion de 'couleur' n'a pas de signification pour VAPS si elle n'est pas associée à un objet. ALEX est plus flexible sur ce point. C'est pourquoi nous avons choisi de nommer les objets de VAPS à partir de la juxtaposition du nom et de l'attribut d'un objet d'ALEX. Ainsi l'objet WATER et l'attribut LEVEL d'ALEX forment l'objet WATERLEVEL de VAPS. La valeur de l'attribut d'ALEX devient l'état de l'objet VAPS.

Prenons l'exemple des trains présenté en annexe C. Dans l'éditeur d'objets de

VAPS, un train est défini à l'aide d'un objet *Lumière*[3], la valeur de l'attribut d'ALEX (la position du train) devient l'état de la lumière correspondante. Le programmeur des règles et le concepteur de l'interface doivent aussi s'entendre sur les valeurs de certaines constantes définies dans le fichier des noms. Considérons le fait PUMP STATE ON (tiré de l'exemple de la mine, voir annexe 1) La constante ON, une fois interprétée par le moteur d'inférence, devient une valeur et celle-ci est passée à l'interface graphique pour devenir un état de l'objet PUMPSTATE, d'où l'importance que la valeur de la constante définie dans le fichier des noms corresponde bien à la valeur de l'état souhaité.

Deux fichiers de canaux doivent être définis pour indiquer le nom des membres des différents éléments de l'interface. Les figures 25 et 26 présentent les fichiers de canaux pour l'exemple de la mine. Ces trois aspects (rigueur des noms et valeurs des constantes et fichiers de canaux) sont les seuls aspects restrictifs reliés à l'interaction entre le moteur parallèle et l'interface graphique.

Affichage partiel des résultats d'inférence : Il va de soi que certains faits inférés par les différents moteurs d'inférence n'ont pas une incidence graphique. Pour limiter le transfert de ces faits sans incidence graphique entre la machine parallèle et l'interface graphique, nous utilisons un fichier de routage similaire à ceux utilisés pour limiter la communication des faits entre les processeurs. Ce fichier est passé en paramètre au programme s'exécutant sur la machine SUN et servant de liens entre ALEX et VAPS.

[3]L'objet lumière de VAPS a une définition plus vaste que celle généralement reconnue. Une lumière VAPS peut être tout objet graphique auquel on peut associer un ou plusieurs états. Chaque train de notre problème est représenté à l'aide d'une lumière à 32 états, chaque état représentant un train à une position différente sur la voie ferrée. En changeant l'état de la lumière, on crée l'impression que le train se 'déplace' sur la voie ferrée.

Contrôles spécifiques : Il est parfois utile de pouvoir agir sur le déroulement de l'exécution d'un programme, nous utilisons alors ce que nous appelons *contrôles spécifiques*. Nous avons démontré que la communication entre ALEX et VAPS pouvait être bidirectionnelle en établissant un mécanisme d'interaction entre l'interface VAPS et la machine parallèle. L'exemple de la mine illustre ce fait à l'aide de trois contrôles agissant respectivement sur la capacité de pompage, le débit d'arrivée du méthane et le débit d'entrée d'eau. Chaque contrôle, en réponse aux modifications apportées par l'utilisateur, dépose une valeur dans son membre associé. Malheureusement, la lecture des membres est indistinctive, c'est-à-dire que nous ne disposons pas de primitives pouvant déterminer si un *membre* contient une nouvelle valeur, mais nous disposons de primitives pour savoir si un *canal* a changé de valeur. VAPS n'offre pas de mécanismes permettant d'être plus sélectif. Nous avons donc résolu le problème en maintenant constamment une table des valeurs courantes des différents membres. Une routine boucle sur ces valeurs et les compare au contenu des membres. Si la comparaison révèle une différence, alors la table des valeurs courantes est mise à jour et un message est produit et envoyé aux différents processeurs mentionnés dans le fichier de routage.

L'exécution sur un processeur éloigné : L'exécution de VAPS sur une machine différente de celle qui exécute le programme de communication entre VAPS et ALEX démontre que la proximité des processus n'est pas nécessaire. Toutes les communications se font via les ports TCP/IP et de ce fait, les processus peuvent s'exécuter sur des machines situées à de grandes distances.

Nous limitons ici notre intervention au niveau de l'affichage et du contrôle de l'interface graphique. Nous invitons le lecteur désireux d'en connaître plus sur le sujet à consulter les différents guides de référence du logiciel VAPS ainsi que le document 'Réalisation d'interfaces graphiques pour un contrôleur de procédés industriels' [Pal95]. Ce dernier document traite de la réalisation d'interfaces graphiques en général mais aussi de la

réalisation spécifique des deux interfaces requis pour le problème du drainage d'une mine et de la circulation des trains. Référez-vous aux annexes B et D pour apprécier le résultat final du travail.

2.3.2 Les messages inter-processeurs et inter-processus

ALEX offre toute une panoplie de primitives de communication par messages inter-processeurs ou inter-processus, toutes ont leurs avantages et leurs inconvénients. Nous présenterons brièvement chacune de ces fonctions dans cette section, puis nous élaborerons sur le choix que nous avons fait et les raisons qui le motivent. Le tableau 3 présente les différentes primitives fournies par ALEX. Chaque série est associée à un niveau d'abstraction différent.

Tableau 3: Tableau des types de messages d'ALEX

Niveau	Primitives
Réseau	n_send, n_recv, n_try_send, n_try_recv
Transport	t_send, t_recv
Lien	d_send, d_recv,d_try_send, d_try_recv
Physique	p_send, p_recv
Noyau	k_send, k_recv, k_try_send, k_try_recv
Local	kpar_He, kpar_Na, kpar_Fe

Il faut ajouter à cette liste les primitives *bsend, brecv, csend* et *crecv* que l'on peut qualifier de *sucre syntaxique*[4], car elles n'offrent aucune fonctionnalité accrue. Elles ne font qu'appeler d'autres fonctions en utilisant des valeurs par défaut pour certains paramètres.

Les primitives de niveau réseau sont les plus simples à utiliser, mais elles sont les plus

[4]Le sucre syntaxique est un terme que l'on associe aux primitives d'un langage lorsque celles-ci permettent d'accéder plus facilement à d'autres primitives.

coûteuses au point de vue traitement. On doit fournir à la fonction *n_send* le nom du processeur destinataire. Par contre, on ne peut pas fournir aux fonctions *recv* le processeur source, donc le premier message arrivé à destination est consommé en premier. Notons que les fonctions *n_send* et *n_try_send* ont toutes deux une approche 'politique' à leurs messages. Pour ces fonctions, *'Un problème repoussé est un problème réglé'*. En effet, si un processeur réussit à se débarrasser d'un message en l'envoyant à son voisin, alors les fonctions *send* considèrent que l'envoi a fonctionné. En d'autres termes, la valeur de retour de la fonction *n_send* indique que le processeur n'a plus le message, mais ceci ne permet en aucun cas d'affirmer que le message a été bien reçu par son destinataire.

Les fonctions *t_send* et *t_recv* assurent une synchronisation inter-processus, car elles n'existent pas en saveur non bloquante. Un processeur destinataire doit faire un *t_recv* pour que le processeur source puisse se libérer de son *t_send*. Au niveau transport, le processeur source doit identifier le destinataire et vice versa.

Les niveaux réseau et transport ont tous deux un mécanisme indépendant et transparant de morcellement des messages en paquets, mais seuls les messages longs (nécessitant morcellement) utilisant les primitives de niveau transport sont assurés d'arriver dans l'ordre au destinataire. Si plusieurs messages longs sont envoyés par plusieurs processeurs avec les primitives *n_send*, alors les messages peuvent arriver non seulement dans le désordre mais aussi entrelacés!

Les primitives de niveau liens n'offrent pas de morcellement automatique des messages longs, et les messages ne peuvent être envoyés qu'à un processeur voisin (dans le sens du schéma de topologie chargé par le fichier *bnail*). De plus, on doit fournir aux fonctions *d_send, d_recv, d_try_send* et *d_try_recv* l'identification du lien sur lequel on désire envoyer le message. L'avantage de ses fonctions est le coût minime associé à l'envoi d'un message.

Au niveau physique, les primitives *p_send* et *p_recv*, comme le nom du niveau l'indique, accèdent directement à la mécanique des liens de communication entre les processeurs. Elle ne peuvent être utilisées qu'après avoir établi un circuit virtuel à l'aide des primitives d'un niveau supérieur. Ces fonctions n'offrent aucune protection contre les erreurs.

Les primitives du niveau noyau sont utilisées pour permettre la communication entre différents processus s'exécutant sur un même processeur; d'ailleurs, c'est par l'entremise de ces primitives de niveau noyau que le système d'exploitation Trollius commande l'exécution de ses diverses tâches.

Finalement, le niveau local permet lui aussi la communication inter-processus sur un même processeur mais cette fois par l'entremise de la création même des processus. Les fonctions *kpar* permettent la création de processus avec un niveau variable de piles mémoire partagée. *Kpar_He* (pour hélium) permet la création de processus 'légers' ne disposant pas de leurs propres piles mémoire. Chaque processus créé avec *kpar_He* partage complétement l'espace mémoire du processus qui l'a créé, il partage aussi ses données. Similairement, un programme exécuté, avec la commande *kpar_Na* (sodium, plus lourd) est un processus ayant sa propre pile mais partageant les données de son créateur. Un processus *kpar_Fe* (fer) est complètement autonome avec sa pile mémoire et son espace de données.

Devant cette multitude de choix de types de communication par message, nous avons d'emblée éliminé les fonctions de niveau local, noyau et physique, car une gestion adéquate de celles-ci nous aurait éloigné de notre but de créer un moteur d'inférence parallèle interagissant avec un logiciel d'interface graphique. Les fonctions de niveau transport n'offrant pas de fonctions non bloquantes sont rejetées, car nous ne devons pas arrêter le moteur

parce qu'un processeur n'est pas prêt à recevoir un message. Nous avons finalement opté pour les fonctions de niveau réseau pour ne pas avoir à tenir compte de la topologie sous-jacente (ce que nous aurions dû faire en utilisant les fonctions de niveau lien). Ce choix est aussi guidé par les auteurs du livre de référence d'ALEX.[5]

Toutefois, nous n'utilisons pas directement les fonctions *n_send, n_recv, n_try_send* et *n_try_rec*, nous utilisons les fonctions *jsend*, *jrecv* et *jtry_recv* (du sucre syntaxique) de la librairie *libjd.* [6]

2.3.3 Structure d'un message de niveau réseau

Nous décrivons dans cette section la structure d'un message de niveau réseau (voir figure 13). Chaque fait qu'un processeur envoie à un autre processeur est encodé dans une structure de message. Le premier champ d'un message est le *nh_dl_event.* Ce champ est utilisé par les mécanismes de communication de niveau lien et physique pour indiquer l'événement qui permet la communication. Puisque nous utilisons les fonctions *n_send, n_recv* et *n_try_recv* nous ignorons par conséquent ce champ.

Le champ *nh_node* est utilisé par le processeur source pour indiquer le numéro du processeur destinataire. Le processeur destinataire n'utilise pas ce champ.

Le *nh_event* contient un 'événement'. Il s'agit en réalité d'un entier positif de 32 bits permettant la synchronisation. Les deux processeurs doivent indiquer le même *nh_event*, sinon il ne peut y avoir de synchronisation.

[5] En page 15 de ALEX-Trollius C Reference [Ale92a] on peut lire: 'By sticking to the network sub-level [primitives] and ignoring half the fields in the network message descriptor, the user will be dumb, happy and productive.'

[6] La librairie *libjd* a été développée en collaboration avec David Jean lors d'un travail d'équipe dans le cadre d'un cours de deuxième cycle.

Le *nh_type* est aussi un entier utilisé pour permettre la synchronisation, mais cette fois l'entier est utilisé comme un champ de 'bits'. Si le *nh_type* est égal à zéro (comme dans notre cas), alors la synchronisation peut s'effectuer avec tous les messages ayant le même *nh_event*. Si par contre le *nh_type* est plus grand que zéro, alors la synchronisation n'est permise qu'avec les messages possédant un *nh_event* identique et un *nh_type* partageant au moins un 'bit' à '1' ; en d'autres termes, le résultat de l'opération 'et logique' des deux *nh_types* doit être différent de zéro.

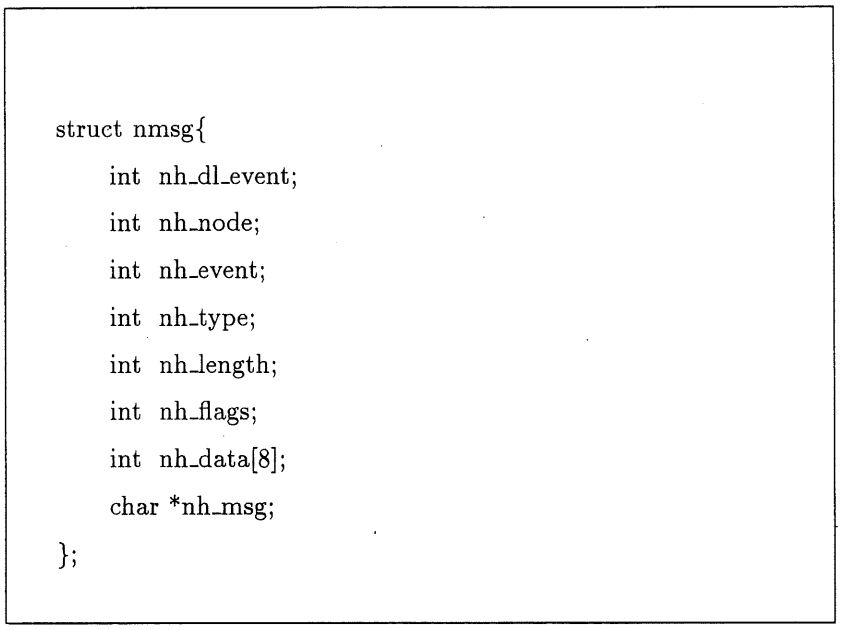
Le *nh_length* indique la longueur du message transféré. Puisque nous n'utilisons pas la partie message, alors cette longueur est nulle pour notre moteur.

Le champ *nh_flags* est un entier utilisé lui aussi comme champ de 'bit' et sert à indiquer différentes options. Ces options servent à indiquer si les mécanismes de communication d'ALEX-Trollius doivent utiliser de tampons. Elles renseignent aussi sur le type de données contenues dans la partie message. Nous n'utilisons pas ce champ de la structure d'un message.

Vient ensuite le champ (tableau) *nh_data[8]*[7]. C'est ici que nous déposons tous les entiers nécessaires pour représenter un fait des bases de faits. Nous utilisons quatre indices du tableau *nh_data* pour passer un message, soit respectivement un pour le *mutateur, l'objet, l'attribut* et un pour la *valeur*.

Comme le lecteur peut le constater, la structure de message *nmsg* offre une grande flexibilité, même en ignorant un bonne partie des champs (nh_dl_event, nh_type, nh_length,

[7]Un programmeur averti n'utilise jamais *nh_data[7]*, car cet indice est utilisé par un outil de traçage de programme nommé Paragraph. Même si nous n'utilisons pas cet outil, nous n'avons pas utilisé cet indice afin de ne pas compremettre la possibilité de traçage lors d'éventuels développements futurs.



```
struct nmsg{
    int  nh_dl_event;
    int  nh_node;
    int  nh_event;
    int  nh_type;
    int  nh_length;
    int  nh_flags;
    int  nh_data[8];
    char *nh_msg;
};
```

Figure 13: Structure d'un message de niveau réseau

nh_flags et *nh_msg). Il demeure possible d'utiliser les autres champs pour permettre une communication simple et efficace entre les processeurs.

2.4 Représentation interne

Notre représentation interne utilise uniquement des nombres entiers. Tous les faits, règles, objets, attributs, valeurs, constantes sont transformés en entiers de 32 bits. Comme nous l'avons vu à la section 2.3.3 portant sur la structure d'un message, nous n'utilisons pas la partie pointeur d'un message proprement dit. Nous nous limitons à la section de contrôle et à un tableau d'entiers. C'est pourquoi nous devons tout traduire en entiers.

L'utilisation d'entiers : Un avantage marqué de cette utilisation a trait à la performance.[8] L'utilisation exclusive d'entiers permet la surcharge d'informations de

[8]Nous avons déjà mentionné à la section des heuristiques 2.2.5 que nous ralentissions artificiellement les processeurs pour permettre une désynchronisation et pour permettre à l'interface d'avoir le temps de

contrôle aux valeurs affectées pour les différents objets, attributs et valeurs. Grâce à cette surimposition, il nous est possible de ne pas utiliser la partie *donnée* des messages d'ALEX-Trollius. Les messages ne sont donc formés que de la section habituellement réservée au *contrôle*. Un message est donc reconnu comme étant de longueur nulle. Un tel message ne requiert aucune allocation dynamique d'espace pour le processeur récepteur, ce qui améliore la performance. Les valeurs de contrôle surimposées aux valeurs réelles nous renseignent sur la sémantique à associer à ces dernières.

Les chaînes de caractère : Le programme ne permet aucune chaîne de caractère dans ses règles. Tout élément d'une règle doit être soit un identificateur, un mot clé ou un constante.

Types, contrôles et indices : ALEX utilise des entiers de 32 bits. Nous avons décidé d'utiliser les neuf derniers bits à des fins de contrôle. Le tableau 4 présente les nombres de contrôle de type utilisés par notre modèle.

Le tableau 5 représente les valeurs minimales associées à chaque liste. À l'étape de lecture du fichier, chaque identificateur présent dans le fichier des noms est transformé et conservé sous forme d'entier à l'aide de la formule suivante :

Entier représentatif = Indice dans la liste + valeur minimal de la liste + constante de type

Pour illustrer l'utilisation des ces différentes constantes et indices, prenons un extrait de la figure 15

ATTRIBUT STATE LEVEL ALARM DANGER SAFE

INPUT OUTPUT BREAK MAX FIN_TABLEAU

consommer ces messages. Aussi prétendre que l'utilisation d'entiers permet d'augmenter la performance peut paraître paradoxal. Néanmoins, nous avons tenté d'optimiser nos processus là où nous le pouvions.

Tableau 4: Tableau des constantes de type

Type	Valeur	Interprétation
THE_SIGN_BIT	16777216	2**24
OBJET	33554432	2**25
ATTRIBUT	67108864	2**26
FONCTION	100663296	2**25 + 2**26
VARIABLE	134217728	2**27
CONSTANTE	167772160	2**27 + 2**25
VALEUR	201326592	2**27 + 2**26
ATOME	234881024	2**27 + 2**26 + 2**25
REGLE	268435456	2**28
PROCESSEUR	301989888	2**28 + 2**25
MUTATEUR	335544320	2**28 + 2**26
PRIORITE	369098752	2**28 + 2**26 + 2**25
VAL_CONSTANTE	402653184	2**28 + 2**27
VALUE_MASK	1107296255	
TYPE_MASK	1040187392	

Tableau 5: Tableau des constantes des indices

Type	Valeur
Min objet	1000
Min attribut	2000
Min fonction	3000
Min variable	4000
Min constante	5000
Min atome	6000
Min règle	7000
Min processeur	8000
Min mutateur	9000
Min priorité	10000

Il s'agit de la liste des attributs dans le fichier des noms. Considérons le cas du mot DANGER, il est le quatrième de la liste (on ne tient pas compte de l'identificateur de la liste en l'occurence du mot ATTRIBUT), donc DANGER est à l'indice trois (3). Le minimum pour la liste des attributs est 2000, selon le tableau 5, et à cela nous devons additionner la valeur de la constante ATTRIBUT (67108864). On obtient alors la représentation interne suivante :

DANGER = 3+2000+67108864 = 67110867

Une série de fonctions de type *constructeur et sélecteur* permettent d'accéder rapidement à la valeur voulue.

Limites des valeurs permises : Notre modèle est limité par la mémoire disponible sur chaque processeur. Aussi nous établissons les limites suivantes pour les différents aspects de nos programmes.

- Le nombre limite de routes par processeur est de 200.
- Le nombre maximum de faits par règle est de 200.
- Il ne peut y avoir plus de 100 règles par processeur.
- Chaque règle ne peut avoir plus de 30 tuples.
- Aucun identificateur ne peut excéder 60 caractères.
- Aucune valeur ne peut excéder 2**24 -1 ou être inférieure à -2**24+1

Domaine d'application

Au moyen de notre modèle, le type de connaissances et les types de raisonnement que l'on peut représenter sont les suivants.

Types de connaissances : La plupart des types de connaissances présentés à la section 1.1.1 peuvent être modélisés avec notre modèle. Toutefois certains concepts pourraient s'avérer plus difficiles que d'autres à modéliser, notamment les concepts incluant une notion de temps réel.

Types de raisonnements : Notre modèle raisonne sur la base du monde clos, il part du principe que tout ce qu'il connaît constitue l'univers. Il n'est pas possible présentement d'inférer de nouveaux faits basés sur des objets ou attributs non présents dans le fichier des noms. Il ne peut pas non plus inférer sur l'inexistence d'un fait. L'ajout de la négation pourrait représenter une avenue de recherche future.

Chapitre 3

Exemples de problèmes

Dans ce chapitre, nous présentons deux exemples de problèmes réalisés à l'aide de notre moteur d'inférence. L'un traite du contrôle d'une pompe de drainage du contenu d'un puits d'une mine, l'autre du déplacement de quatres trains sur des voies ferrées. Les annexes A et B présentent respectivement les fichiers de paramètres et quelques images du problème de la mine. Les annexes C et D présentent les fichiers de paramètres et quelques images du problème des trains.

3.1 Le drainage d'une mine

Le problème de drainage d'une mine est un problème de contrôle. Trois objets doivent interagir selon certaines règles élémentaires. La situation est la suivante : Une mine possède un puits de drainage où s'accumule toutes les infiltrations d'eau. Une pompe vidange le puits de drainage. Un capteur détecte la présence de méthane ; deux autres capteurs détectent le niveaux minimum et maximum d'eau.

La contrainte suivante doit être respectée : la pompe devra être mise en marche si et seulement si le niveau d'eau dans le puits de drainage le requiert et si la présence de

méthane est à un niveau acceptable. Pour une description complète du problème, voir [Bar94].

Les différents fichiers de paramètres requis pour permettre l'exécution du problème sont présentés en annexe 1. La figure 15 montre les sept listes contenant les *noms* du problème. Le problème est décrit à l'aide de trois objets : *la pompe, l'eau et le méthane.* Les attributs STATE, BREAK ET OUTPUT sont associés à la pompe, les attributs LEVEL, ALARM, DANGER, SAFE ET INPUT sont associés au méthane et à l'eau. La liste des fonctions prédéfinies est présentée à la liste FONCTION. Les variables requises par les règles sont énumérées dans la liste VARIABLE. Les constantes et leurs valeurs associées (selon la méthode d'association définie à la section 2.2.1) sont présentées dans les listes CONSTANTE et VAL_CONSTANTE. Finalement, la liste REGLE contient les noms de chacune des règles.

La figure 16 montre quelles sont les règles qu'utilise le responsable de la pompe (processeur P1). Nous décrivons maintenant chacune des règles du processeur P1.

La règle PUMP_ON : Cette règle permet d'activer la pompe si celle-ci est présentement arrêtée, si elle n'est pas brisée, si l'alarme d'eau est activée et s'il n'y a pas d'alarme de méthane. Le règle PUMP_ON est la seule règle qui contrôle l'activation de la pompe.

La règle PUMP_OFF : Ordonne l'arrêt de la pompe si l'alarme d'eau est activée.

La règle PUMP_OFF_2 : Arrête la pompe si le niveau de méthane devient critique.

La règle PUMPING : Tant et aussi longtemps que la pompe sera activée, cette règle enverra un message au processeur responsable du niveau d'eau l'invitant à réduire ce niveau de la valeur correspondant au débit de la pompe.

La règle BREAKDOWN : Arrête la pompe dès qu'il y a un bris d'équipement.

Le processeur P1 gère donc cinq règles : une pour activer la pompe, trois pour arrêter la pompe et une pour le pompage. La règle de pompage a une priorité PS0, ce qui signifie qu'elle sera activée si ses préconditions le permettent, mais aussi si aucune autre règle n'est activable. Les règles PUMP_OFF et PUMP_OFF_2 ont toutes deux une priorité statique PS2. Finalement, la règle BREAKDOWN est la règle la plus prioritaire statiquement avec PS4 ; elle sera donc exécutée avant toute autre si sa condition d'activation devient satisfaite.

La figure 19 présente les faits initiaux connus du processeur P1. Comme nous pouvons le constater, les faits initiaux du processeur responsable de la pompe ne lui permettent aucune inférence. En conséquence, le moteur d'inférence sera mis en attente de messages modifiant sa base de faits et ainsi lui permettant d'entrer dans un cycle d'évaluation.

La figure 17 contient les trois règles qu'utilise le responsable de l'eau (processeur P2). Élaborons sur chacune de ces règles.

La règle INPUT_WATER : Cette règle augmente le niveau d'eau si aucune autre règle n'est activable.

La règle WATER_ALARM_ON : Comme son nom l'indique, elle active l'alarme d'eau si le seuil critique est atteint.

La règle WATER_ALARM_OFF : Inversement à la règle précédente, cette règle désactive l'alarme d'eau si le seuil sécuritaire est atteint.

La figure 20 présente les faits initiaux connus du processeur P2, responsable de l'eau. Sa base de faits initiaux est très simple : le niveau courant de l'eau, le débit d'arrivée d'eau et l'état de l'alarme d'eau. Les valeurs des seuils sécuritaires et critiques sont des constantes contenues dans le fichier des noms (figure 15). Les faits initiaux permettent au moteur d'inférer la règle INPUT_WATER mais le fait WATER INPUT 0 limite

son effet, car il ne permet aucune modification de la base de faits. Le moteur effectue selon son cycle l'évaluation de cette règle jusqu'à ce qu'un message lui permette autre chose.

La figure 18 contient les trois règles qu'utilise le responsable du méthane (processeur P3). Les règles régissant le méthane sont les mêmes que celles gérant l'eau ; seul les mots utilisés diffèrent.

La règle INPUT_METHANE : Cette règle augmente le niveau de méthane si aucune autre règle n'est activable.

La règle METHANE_ALARM_ON : Active l'alarme de méthane si le seuil critique est atteint.

La règle METHANE_ALARM_OFF : Désactive l'alarme de méthane si le seuil sécuritaire est atteint.

La figure 21 présente les faits initiaux connus du processeur P3, le processeur responsable du méthane. Tout comme la base de faits relative à l'eau, la base de faits initiale pour le méthane permet l'activation d'une règle qui ne produit aucun changement dans la base de faits. Le moteur d'inférence qui gère le méthane est lui aussi en boucle continue d'évaluation inutile.

Le *fichier des routes* est présenté à la figure 22. Ce fichier est commun à tous les processeurs sauf au processeur *racine*[1]. Le processeur racine porte le nom P100 dans figure 22.

La figure 23 contient les informations du *fichier des routes pour* VAPS, ce fichier est plus succint que le fichier de routage des autres processeurs, car il ne contient que les

[1]Le processeur *racine* est physiquement situé sur une carte dans le serveur SUN, c'est lui qui permet le lien avec d'autres applications que celles s'exécutant sur les processeurs de la machine ALEX.

informations ayant trait à l'interface VAPS. Finalement, le *fichier des faits initiaux de VAPS* est présenté à la figure 24.

3.2 Les trains

Le problème des trains est aussi un problème de contrôle. Mais cette fois-ci il s'agit de quatre trains qui circulent sur de nombreuses voies ferrées. Chaque voie est divisée en sections. Des lumières de circulation contrôlent le mouvement des trains. Un train ne peut avancer que si la lumière de la voie devant lui est verte. Lorsqu'un train ne peut avancer sur une voie, il tente d'avancer sur une autre, le but étant de faire circuler tous les trains simultanément sans collision.

3.2.1 Modélisation

Chaque train est modélisé à l'aide d'une lumière à 32 états. Chaque état de la lumière représente le train sur un segment de la voie ferrée. Les voies ferrées sont représentées à l'aide de sections et certaines sections sont subdivisées en segments. À l'extrémité de chaque section, on retrouve un lumière rouge ou verte selon que la section est occupée ou non par un train. Notre modèle comprend quatre trains, un rouge, un bleu, un jaune et un vert.

3.2.2 Gestion du modèle

Chaque train circule conformément à son fichier de faits initiaux. Dans ce dernier, on retrouve des faits comme

S1 NEXTSECTOR S2

S2 NEXTSECTOR S4

Ces faits indiquent l'ordre de déplacement du train entre les sections. Pour un autre train, le fait :

S1 NEXTSECTOR S7

indique un chemin différent de celui emprunté par le train précédent. Conceptuellement, on peut penser qu'une section de voie ferrée est composée de plusieurs segments et qu'elle est délimitée par deux lumières, mais à l'interne il n'y a que des sections. C'est donc l'attribut *NEXTSECTOR* qui permet de gérer le sens de déplacement ainsi que les voies utilisées. L'attribut *NEXTSECTOR* indique l'ordre de déplacement des trains. Chaque section (et non segment) de voie ferrée est délimitée par deux lumières. Si les lumières de sections sont au rouges, alors cela signifie que la section est occupée par train. Une section ne peut être occupée que par un seul train à la fois. Nous avons voulu rendre le problème plus complexe en forçant chaque train à vérifier l'état des lumières de sections qu'il désire occuper plutôt que de regarder la position des trains sur les sections, ce qui aurait été plus simple. Le déroulement des inférences suit l'algorithme de la figure 14.

C'est ainsi que chaque train peut se déplacer d'un segment à un autre et d'une section à une autre sans entrer en collision avec un autre train. Notons que la règle qui gère les déplacements d'un segment à un autre a une priorité statique supérieure à la règle de déplacement inter-sections. Cette priorité statique assure des déplacements logique dans le cas où les deux règles sont instanciables.

Pour l'exemple du train nous n'avons pas construit d'interface permettant à l'usager d'intervenir directement. Si un usager désire arrêter un train à un segment particulier, il doit intervenir au niveau du fichier des faits en coupant la continuité des déplacements. L'annexe 3 présente les fichiers de paramètres et l'annexe 4 illustre le problème des trains à différentes étapes de son exécution.

```
La section de voie ferrée actuelle a-t-elle plusieurs segments?
Si oui
    Le train est-il situé sur le dernier segment de la section?
    Si oui
        Quel est la prochaine section?
        Les lumières de la prochaine section sont-elles vertes?
        Si oui
            Mettre les lumières de la prochaine section au rouge
            Avancer le train au premier segment de la prochaine section
            Mettre les lumières de l'ancienne section au vert
        Si non
            Attendre
    Si non
        Avancer au prochain segment
Si non
    Quel est la prochaine section?
    Les lumières de la prochaine section sont-elles vertes?
    Si Oui
        Mettre les lumières de la prochaine section au rouge
        Avancer le train au premier segment de la prochaine section
        Mettre les lumières de l'ancienne section au vert
    Si non
        Attendre
```

Figure 14: Algorithme de déplacement d'un train

Conclusion

Objectifs

Notre but était de concevoir et développer un moteur d'inférence parallèle pouvant résoudre un problème parallélisé sur un multiprocesseur. Ainsi, nous devions établir un lien de communication entre la machine parallèle ALEX et le logiciel d'interface graphique VAPS. Nous avons démontré par nos recherches et par les résultats présentés ici que nous avons atteint nos objectifs. Effectivement, un problème parallélisé peut être réparti sur différents processeurs agissant sur leur base de faits respective et communiquant entre eux via des primitives de communication par messages. Nous avons aussi démontré que la communication entre la machine parallèle et le logiciel d'interface graphique est non seulement possible mais, qui plus est, bi-directionnelle. Nous croyons que plusieurs approches originales et non triviales furent mises de l'avant dans ce projet. L'utilisation exclusive d'entiers comme représentation interne nous a permis de minimiser les messages à leur plus simple expression (la partie *message* proprement dite est de longueur nulle). L'amalgame d'heuristiques utilisées dans notre modèle, nous permet d'assurer une grande flexibilité dans la conception des règles tout en permettant à chaque processeur d'avoir éventuellement sa chance de pouvoir exécuter ses règles. Les priorités statiques combinées à un processus de vieillissement des règles instanciables assurent une équité intra-processeur. L'utilisation des faits les plus récents pour réinitialiser la boucle d'évaluation des appariements couplée au paramètre de désynchronisation assurent une

justice inter-processeurs. Rappelons aussi que l'utilisation des fichiers de paramètres prouve la généralité du modèle. Le fichier de routage permet même de limiter le débit des messages inter-processeurs au strict minimum, non seulement en avisant uniquement lorsqu'il y a une modification des *valeurs* des relations *objet-attribut* mais aussi en permettant de ne distribuer ces informations qu'aux processeurs concernés.

Recherches futures

Plusieurs avenues de recherches futures se sont présentées lors du développement de notre moteur d'inférence parallèle. La première avenue de recherche que nous mentionnons n'a pas trait à notre moteur comme tel, mais bien à la grande difficulté de développer ce dernier en l'absence d'un outil de déverminage pour la machine parallèle. À de nombreuses reprises, nous nous sommes dits que développer un tel outil serait un projet de recherche colossal mais stimulant.

Mais revenons à notre moteur d'inférence. Nous avons déjà mentionné que lors de l'évaluation d'une règle, l'évaluation des conséquents de cette règle est perçue comme une opération atomique pour le processeur source, mais comme une suite de faits détachés pour le processeur destinataire. Ce morcellement peut en théorie permettre à un processeur de recevoir une partie des conséquents produits par l'exécution d'une règle d'un autre processeur, effectuer son cycle d'évaluation et inférer une règle avant même d'avoir reçu le reste des conséquents de la règle originale. Les implications associées à ce comportement peuvent être considérables. Des recherches sur ces implications et sur leurs solutions méritent d'être faites.

Des recherches futures pourraient aussi être entreprises pour généraliser notre modèle au monde ouvert en lui permettant d'inférer sur l'absence de faits.

Il serait aussi désirable de rechercher une solution au petit nombre de tampons disponibles lors de la communication entre ALEX et VAPS via les ports TCP/IP. Ce sont ces tampons (et la lenteur de VAPS) qui obligent notre modèle à ralentir artificiellement le déroulement des inférences. Toutefois, augmenter le nombre de tampons pourrait soulever quantités d'autres problèmes non-triviaux. Prenons le scénario suivant. Un problème est parallélisé sur plusieurs processeurs, et l'utilisateur, via l'interface graphique, peut intervenir dans le déroulement du programme. S'il y a *n-1* messages non traités dans les tampons, l'utilisateur dispose d'une vue décalée dans le temps de la base de faits. Il voit la base telle qu'elle était au stade *n-1* inférences. S'il décide d'intervenir dans le déroulement du programme, que se passe-t-il?

- Les faits sur lesquels il désire agir existent-ils encore présentement? Peut-être.
- Son intervention devrait-elle être ajoutée comme *n-ième* message et traitée le moment venu si elle est toujours valide? Et si c'est ce que nous faisons, l'utilisateur après le déroulement des *n-1* messages est-il toujours d'accord avec son intervention? Rien n'est moins sûr.
- L'intervention humaine devrait-elle avoir un impact réinitialisateur? Suite à une intervention de l'usager, notre programme devrait-il ramener ses bases dans l'état où elles étaient au moment de l'intervention? Cette approche serait lourde à gérer et limiterait le domaine d'application à un carcan théorique, car remonter dans le temps relève de la fiction.

Ce problème illustre une fois de plus que le domaine de la recherche est plus vaste que le domaine du connu. C'est notre avis que malgré tous les efforts pour reproduire artificiellement la pensée humaine, seules la curiosité et la persévérance nous permettent vraiment de réduire l'écart entre l'inconnu et le connu.

Annexe A: Le drainage d'une mine : les fichiers de paramètres

Cette annexe présente tous les fichiers de paramètres permettant de définir le problème du drainage d'un puits d'une mine. La figure 15 montre le contenu du fichier des noms du problème. Les listes des objets, des attributs, des fonctions, des constantes, des valeurs de constantes et des règles y sont définies. La figure 16 présente les cinq règles du processeur P1 régissant la pompe. La figure 17 nous indique les trois règles régissant le niveau d'eau et les alarmes de haut niveau et de bas niveau de l'eau. Le processeur P3 s'occupe du méthane à l'aide des trois règles présentées à la figure 18. Notez la grande similitude entre les règles régissant l'eau et celles régissant le méthane. Les figures 19, 20 et 21 présentent les fichiers de faits des processeurs P1, P2 et P3. Les figures 22 et 23 affichent le contenu des fichiers de routage des processeurs et du programme de communication entre ALEX et VAPS. La figure 24 affiche le contenu du fichier des faits pour VAPS, c'est-à-dire quel est l'état initial de l'affichage de l'interface. Finalement, les figures 25 et 26 nous indiquent les noms des divers canaux requis pour permettre la communication bidirectionnelle entre ALEX et VAPS.

```
// ----------------------
// Les trois objets du problème :
OBJET PUMP WATER METHANE FIN_TABLEAU
//
ATTRIBUT STATE LEVEL ALARM DANGER SAFE INPUT
        OUTPUT BREAK MAX FIN_TABLEAU
//
FONCTION Plus Minus Equal Smaller Greater Show Mult Min Max FIN_TABLEAU
//
VARIABLE $level $safe $newlevel $danger $state $input $newinput
        $output $null FIN_TABLEAU
//
CONSTANTE ON OFF WATER_SAFE WATER_DANGER WATER_MAX
        WATER_MIN METHANE_SAFE METHANE_DANGER
        METHANE_MAX METHANE_MIN FIN_TABLEAU
//
VAL_CONSTANTE 2 1 40 440 500 5 40 440 500 5 FIN_TABLEAU
//
REGLE INPUT_WATER INPUT_METHANE PUMP_ON PUMP_OFF
        PUMP_OFF_2 WATER_ALARM_ON WATER_ALARM_OFF
        METHANE_ALARM_ON METHANE_ALARM_OFF PUMPING
        BREAKDOWN FIN_TABLEAU
```

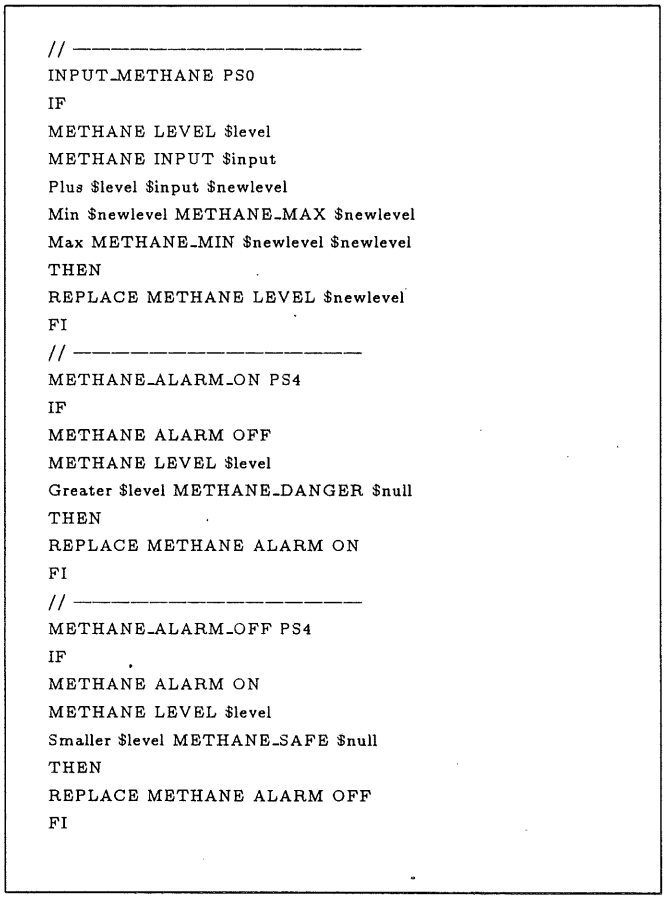
Figure 15: Mine : fichier des noms

```
// ----------------------
PUMP_ON PS2
IF
PUMP STATE OFF
PUMP BREAK OFF
WATER ALARM ON
METHANE ALARM OFF
THEN
REPLACE PUMP STATE ON
FI
// ----------------------
PUMP_OFF PS2
IF
PUMP STATE ON
WATER ALARM OFF
THEN
REPLACE PUMP STATE OFF
FI
// ----------------------
PUMP_OFF_2 PS2
IF
PUMP STATE ON
METHANE ALARM ON
THEN
REPLACE PUMP STATE OFF
FI
// ----------------------
PUMPING PS0
IF
PUMP BREAK OFF
PUMP STATE ON
PUMP OUTPUT $output
THEN
SUBFROM WATER LEVEL $output
FI
// ----------------------
BREAKDOWN PS4
IF
PUMP BREAK ON
THEN
REPLACE PUMP STATE OFF
FI
```

Figure 16: Mine : fichier des règles du processeur P1

```
// ----------------------
INPUT_WATER PS0
IF
WATER LEVEL $level
WATER INPUT $input
Plus $level $input $newlevel
Min $newlevel WATER_MAX $newlevel
Max WATER_MIN $newlevel $newlevel
THEN
REPLACE WATER LEVEL $newlevel
FI
// ----------------------
WATER_ALARM_ON PS4
IF
WATER ALARM OFF
WATER LEVEL $level
Greater $level WATER_DANGER $null
THEN
REPLACE WATER ALARM ON
FI
// ----------------------
WATER_ALARM_OFF PS4
IF
WATER ALARM ON
WATER LEVEL $level
Greater WATER_SAFE $level $null
THEN
REPLACE WATER ALARM OFF
FI
```

Figure 17: Mine : fichier des règles du processeur P2



```
// ------------------------
INPUT_METHANE PS0
IF
METHANE LEVEL $level
METHANE INPUT $input
Plus $level $input $newlevel
Min $newlevel METHANE_MAX $newlevel
Max METHANE_MIN $newlevel $newlevel
THEN
REPLACE METHANE LEVEL $newlevel
FI
// ---------------------
METHANE_ALARM_ON PS4
IF
METHANE ALARM OFF
METHANE LEVEL $level
Greater $level METHANE_DANGER $null
THEN
REPLACE METHANE ALARM ON
FI
// ---------------------
METHANE_ALARM_OFF PS4
IF
METHANE ALARM ON
METHANE LEVEL $level
Smaller $level METHANE_SAFE $null
THEN
REPLACE METHANE ALARM OFF
FI
```

Figure 18: Mine : fichier des règles du processeur P3

```
// ----------------------
// Ce processeur est responsable de la pompe
// État initial de la pompe
PUMP STATE OFF
PUMP BREAK OFF
PUMP OUTPUT 0
// Aucune arrivé de méthane et d'eau
METHANE INPUT 0
WATER INPUT 0
// Aucune alarme activée
WATER ALARM OFF
METHANE ALARM OFF
```

Figure 19: Mine : fichier des faits du processeur P1

```
// ----------------------
// Ce processeur est responsable de l'eau
WATER LEVEL 5
WATER INPUT 0
WATER ALARM OFF
```

Figure 20: Mine : fichier des faits du processeur P2

```
// ----------------------
// Ce processeur est responsable du méthane
METHANE LEVEL 5
METHANE INPUT 0
METHANE ALARM OFF
```

Figure 21: Mine : fichier des faits du processeur P3

```
// ----------------------
PUMP STATE P1
PUMP BREAK P1
PUMP OUTPUT P1
WATER ALARM P1
WATER ALARM P2
WATER INPUT P2
WATER LEVEL P2
METHANE ALARM P1
METHANE ALARM P3
METHANE LEVEL P3
METHANE INPUT P3
PUMP STATE P100
WATER LEVEL P100
METHANE LEVEL P100
```

Figure 22: Mine : fichier des routes pour ALEX

```
// ----------------------
PUMP BREAK P1
PUMP OUTPUT P1
WATER INPUT P2
METHANE INPUT P3
PUMP STATE P100
WATER LEVEL P100
METHANE LEVEL P100
```

Figure 23: Mine : fichier des routes pour VAPS

```
// ----------------------
PUMP BREAK OFF
PUMP OUTPUT 0
WATER INPUT 0
METHANE INPUT 0
PUMP STATE OFF
WATER LEVEL 5
METHANE LEVEL 5
```

Figure 24: Mine : fichier des faits pour VAPS

```
// ---------------------
SCOPE SESSION
TYPE FAST
WATERINPUT    1    F
METHANEINPUT  1    F
PUMPOUTPUT    1    F
PUMPBREAK     1    F
```

Figure 25: Mine : fichier des canaux de sortie pour la mine

```
// ---------------------
SCOPE SESSION
TYPE FAST
PUMPSTATE     1    F
WATERLEVEL    1    F
METHANELEVEL  1    F
```

Figure 26: Mine : fichier des canaux d'entrée pour la mine

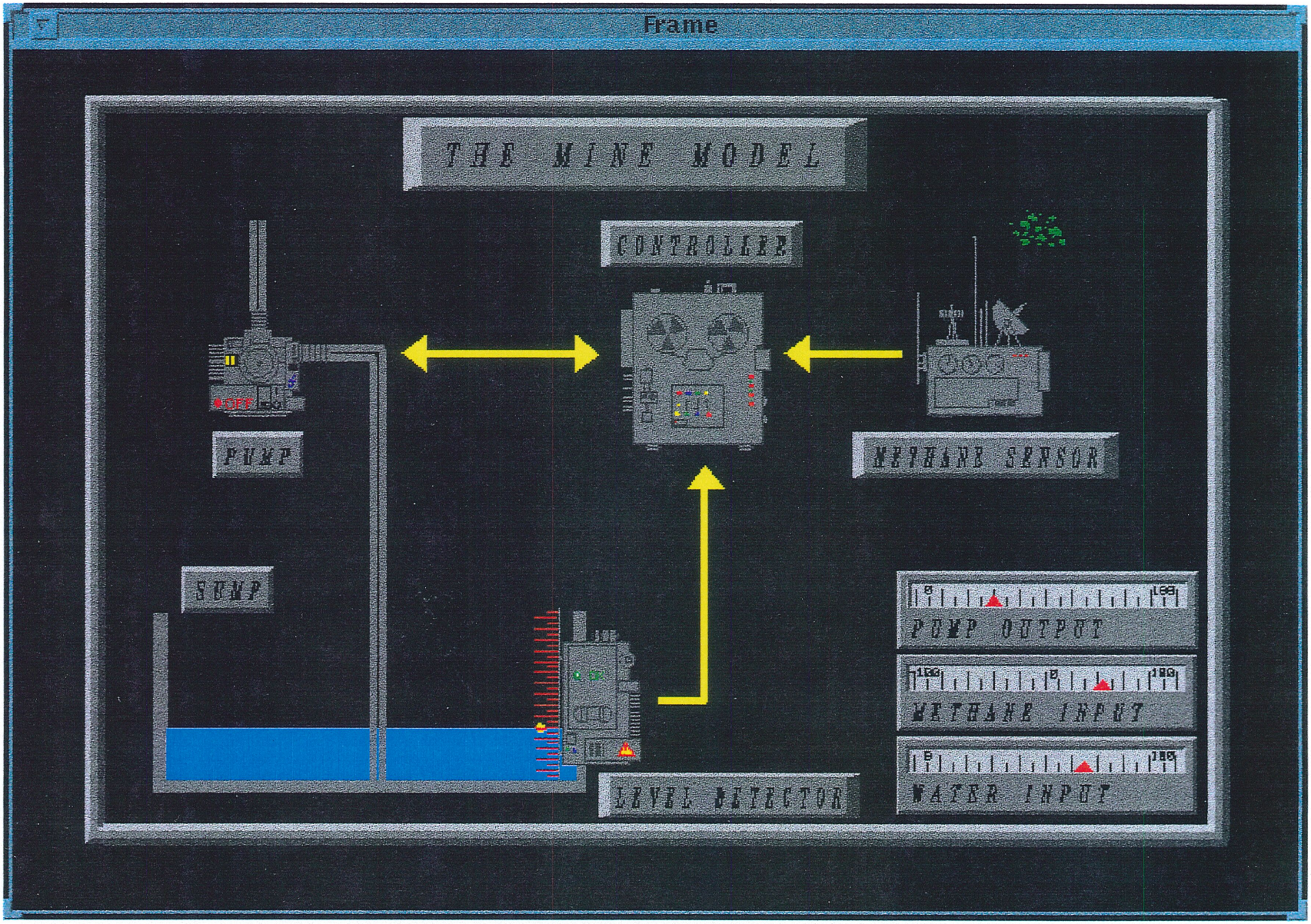
Annexe B: Le drainage d'une mine : les images de l'interface

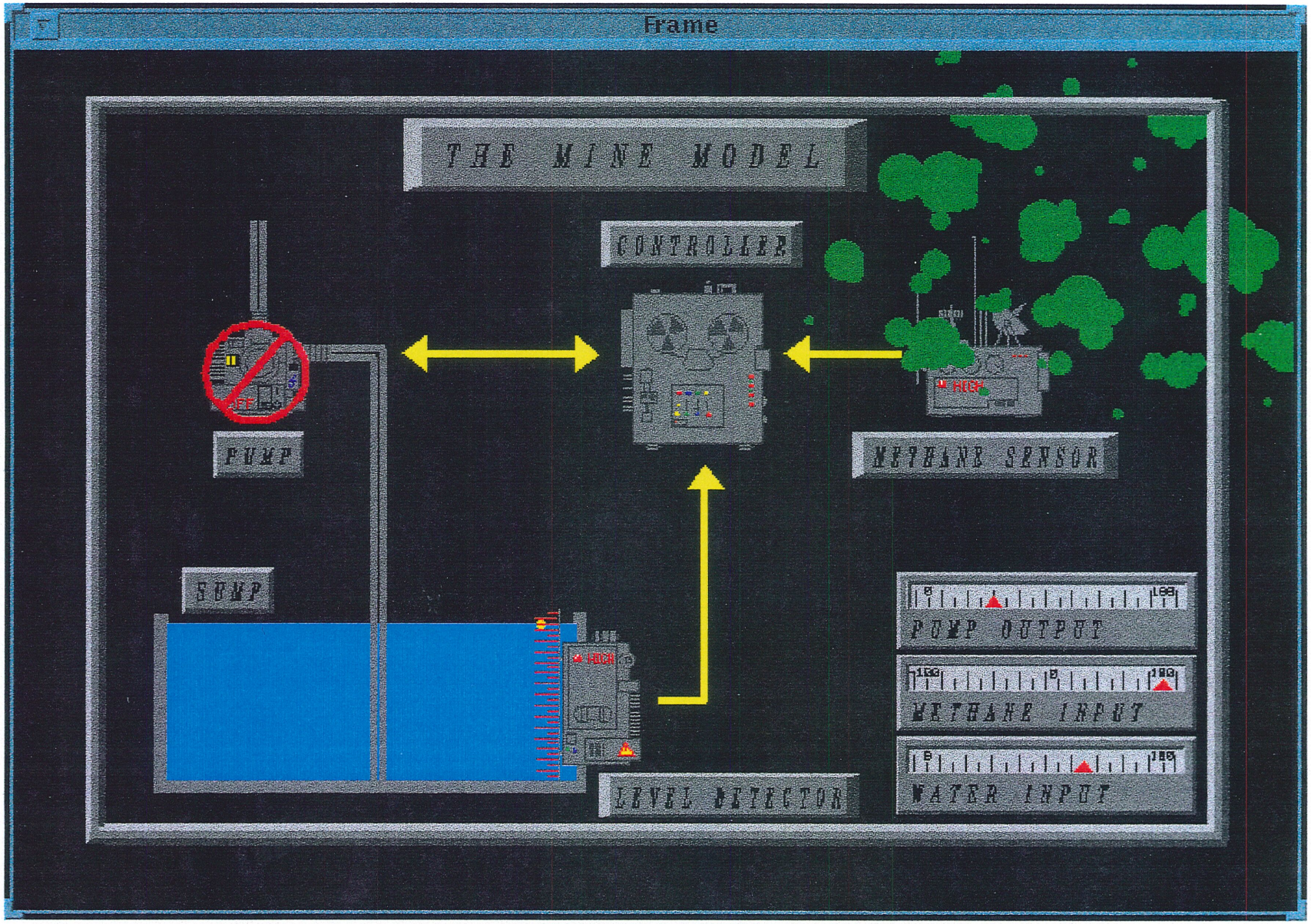
Les trois pages qui suivent présentent le problème de la mine à différents stades d'exécution.

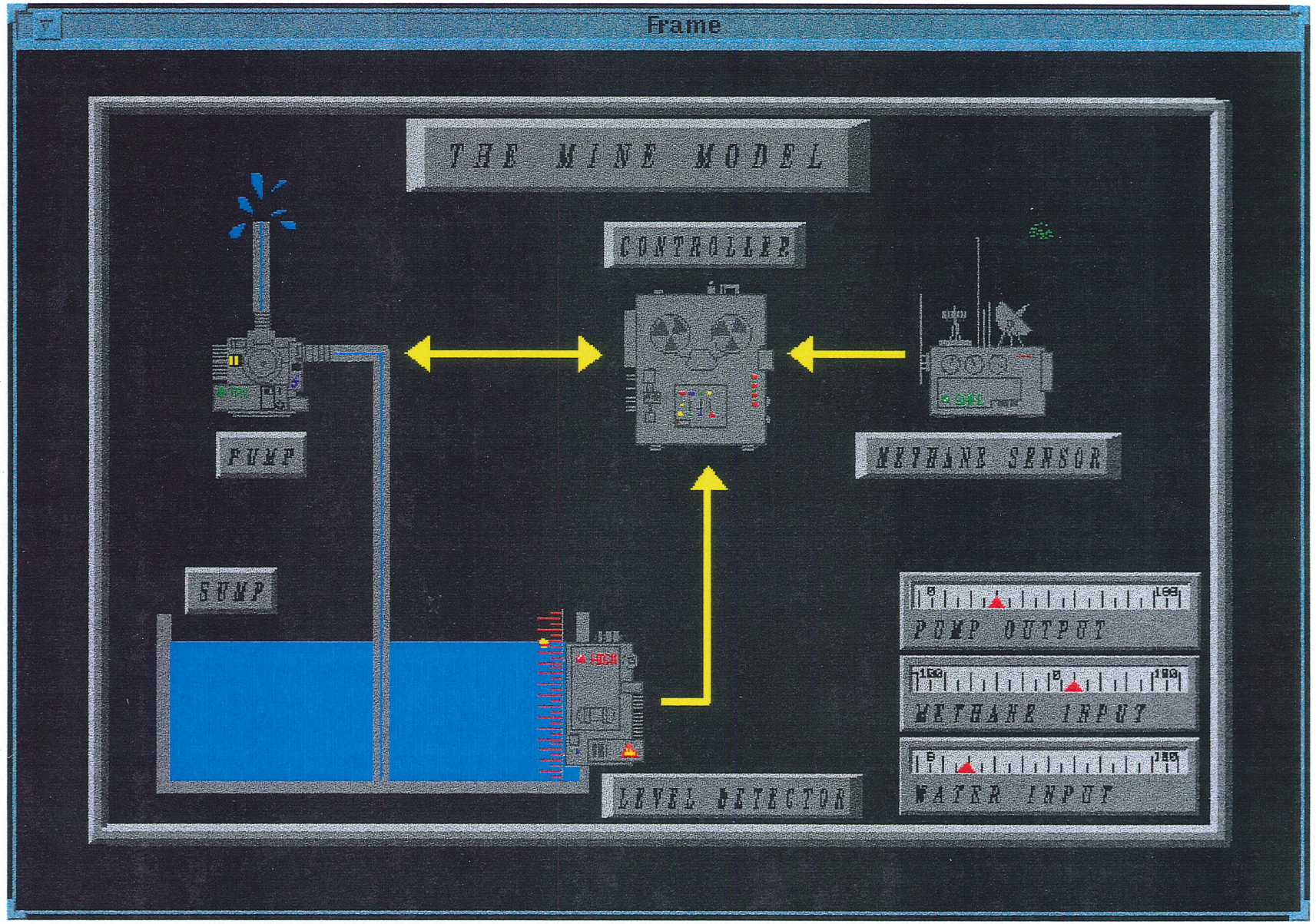
Première image : Sur cette image, la pompe est arrêtée, on détecte la présence de méthane mais pas en quantité suffisante pour déclancher l'alarme. L'eau monte dans le puits au rythme de 65 litres par unité de temps. La concentration de méthane augmente de 22 ppm par unité de temps. Si la pompe était activée, elle débiterait de 30 litres par unité de temps.

Seconde image : Ici, rien ne va plus. L'eau a atteint son seuil critique, l'alarme de méthane a été activée et la pompe est brisée.

Troisième image : Il n'y a que quelques traces de méthane, la pompe est en marche et la vitesse d'arrivée d'eau est inférieure au débit de la pompe.


Frame
THE MINE MODEL
CONTROLLER
PUMP
METHANE SENSOR
SUMP
LEVEL DETECTOR
PUMP OUTPUT
METHANE INPUT
WATER INPUT


Frame
THE MINE MODEL
CONTROLLER
PUMP
METHANE SENSOR
SUMP
LEVEL DETECTOR
PUMP OUTPUT
METHANE INPUT
WATER INPUT


Frame
THE MINE MODEL
CONTROLLER
PUMP
METHANE SENSOR
SUMP
PUMP OUTPUT
METHANE INPUT
WATER INPUT
LEVEL DETECTOR

Annexe C: Le chemin de fer : les fichiers de paramètres

Cette annexe présente les fichiers de paramètres du problème des trains circulant sur des voies ferrées. La figure 27 montre le contenu du fichier des noms du problème. Les listes des objets, des attributs, des fonctions, des constantes, des valeurs de constantes et des règles y sont définies. La figure 28 présente les deux règles contrôlant les mouvements de tous les trains. Tous les processeurs possèdent le même fichier de règles. Les distinctions se manifestent au niveaux des fichiers de faits (figures 29, 30, 31, 32, 33, 34, 35, 36 et 37). Outre le nom du train qui diffère d'un fichier de paramètres à un autre, les distinctions suivantes méritent d'être soulignées. Les trains ne connaissent pas toutes les lumières, certain trains, notamment le train T1, connaît plus de lumières que le train T2. Ceci s'explique par le fait que l'itinéraire du train T1 est plus long que celui du train T2. Notons aussi une autre différence entre les différents fichiers de faits. La valeur de l'attribut NEXTSECTOR d'une même section diffère d'un train à l'autre. Le train T2 affiche le fait S4 NEXTSECTOR S5 tandis que le train T4 possède le fait S4 NEXTSECTOR S26. Il en est ainsi, car ces deux trains circulent en sens inverse l'un par rapport à l'autre. Les figures 38, 39, 40 et 41 composent le fichier de routage des processeurs.

```
// ----------------------
OBJETS T1 T2 T3 T4 L1 L2 L3 L4 L5 L6 L7 L8 L9 L10 L11 L12 L13 L14 L15
     L16 L17 L18 L19 L20 L21 L22 L23 L24 L25 L26 L27 L28 S1 S2 S3 S4 S5 S6
     S7 S8 S9 S10 S11 S12 S13 S14 S15 S16 S17 S18 S19 S20 S21 S22 S23 S24
     S25 S26 S27 S28 S29 S30 S31 S32 NEXT TRAIN FIN_TABLEAU
// "T"n signifie TRAIN numero "n"
// "L"n signifie LUMIERE numero "n"
// "S"n signifie SECTEUR numero "n"
//
ATTRIBUTS POSITION SECTOR VALUE LIGHTA LIGHTB FIRST NEXTSECTOR NAME
     STATE FIN_TABLEAU
//
FONCTIONS Plus Minus Equal Smaller Greater Show Mult Min Max FIN_TABLEAU
//
VARIABLES $currLightA $currLightB $sector $nextsector $sectorvalue $train
     $lightB $lightA FIN_TABLEAU
//
CONSTANTES GREEN RED FIN_TABLEAU
//
VAL_CONSTANTES 2 1 FIN_TABLEAU
//
REGLES TRAIN_SAME_SECTION TRAIN_CHANGE_SECTION FIN_TABLEAU
```

Figure 27: Train : fichier des noms

```
// ----------
// ----------
// ----------
TRAIN_SAME_SECTION PS2
IF
$train SECTOR $sector
$sector LIGHTA $lightA
$sector LIGHTB $lightB
$sector NEXTSECTOR $nextsector
$nextsector VALUE $sectorvalue
$nextsector LIGHTA $lightA
$nextsector LIGHTB $lightB
$lightA STATE RED
$lightB STATE RED
THEN
REPLACE $train SECTOR $nextsector
REPLACE $train POSITION $sectorvalue
FI
// ----------
TRAIN_CHANGE_SECTION PS3
IF
$train SECTOR $sector
$sector LIGHTA $currLightA
$sector LIGHTB $currLightB
$sector NEXTSECTOR $nextsector
$nextsector VALUE $sectorvalue
$nextsector LIGHTA $lightA
$nextsector LIGHTB $lightB
$currLightA STATE RED
$lightA STATE GREEN
THEN
REPLACE $lightA STATE RED
REPLACE $lightB STATE RED
REPLACE $train SECTOR $nextsector
REPLACE $train POSITION $sectorvalue
REPLACE $currLightA STATE GREEN
REPLACE $currLightB STATE GREEN
FI
// ----------
```

Figure 28: Train : fichier des règles

```
// ————————————————————
// ————————-
TRAIN NAME T1
T1 SECTOR S30
T1 POSITION 30
// ————————-
L1 STATE RED
L6 STATE RED
L4 STATE GREEN
L9 STATE GREEN
L8 STATE RED
L21 STATE RED
L7 STATE RED
L10 STATE RED
L12 STATE GREEN
L19 STATE GREEN
L22 STATE GREEN
L23 STATE GREEN
L25 STATE GREEN
L5 STATE GREEN
L11 STATE GREEN
L13 STATE GREEN
L15 STATE GREEN
L3 STATE GREEN
// ————————-
S1 LIGHTA L1
S1 LIGHTB L6
S2 LIGHTA L1
S2 LIGHTB L6
S3 LIGHTA L1
S3 LIGHTB L6
// ————————-
S4 LIGHTA L4
S4 LIGHTB L9
S5 LIGHTA L4
S5 LIGHTB L9
S6 LIGHTA L4
S6 LIGHTB L9
// ————————-
S7 LIGHTA L7
S7 LIGHTB L10
S8 LIGHTA L7
S8 LIGHTB L10
S9 LIGHTA L7
S9 LIGHTB L10
```

Figure 29: Train : fichier des faits relatifs au train T1

```
// ----------------------
// -------.
S10 LIGHTA L11
S10 LIGHTB L13
S11 LIGHTA L11
S11 LIGHTB L13
// -------.
S12 LIGHTA L15
S12 LIGHTB L3
S13 LIGHTA L15
S13 LIGHTB L3
S14 LIGHTA L15
S14 LIGHTB L3
S15 LIGHTA L15
S15 LIGHTB L3
S16 LIGHTA L15
S16 LIGHTB L3
S17 LIGHTA L15
S17 LIGHTB L3
S20 LIGHTA L22
S20 LIGHTB L23
// -------.
S26 LIGHTA L25
S26 LIGHTB L5
S27 LIGHTA L25
S27 LIGHTB L5
// -------.
S28 LIGHTA L19
S28 LIGHTB L12
S29 LIGHTA L19
S29 LIGHTB L12
// -------.
S30 LIGHTA L21
S30 LIGHTB L8
S31 LIGHTA L21
S31 LIGHTB L8
S32 LIGHTA L21
S32 LIGHTB L8
// -------.
S1 VALUE 1
S2 VALUE 2
S3 VALUE 3
S4 VALUE 4
S5 VALUE 5
```

Figure 30: Train : fichier des faits relatifs au train T1 (suite)

```
// ————————————————————
S6 VALUE 6
S7 VALUE 7
S8 VALUE 8
S9 VALUE 9
S10 VALUE 10
S11 VALUE 11
S12 VALUE 12
S13 VALUE 13
S14 VALUE 14
S15 VALUE 15
S16 VALUE 16
S17 VALUE 17
S20 VALUE 20
S26 VALUE 26
S27 VALUE 27
S28 VALUE 28
S29 VALUE 29
S30 VALUE 30
S31 VALUE 31
S32 VALUE 32
// ————————.
S4 NEXTSECTOR S5
S5 NEXTSECTOR S6
S6 NEXTSECTOR S7
S7 NEXTSECTOR S8
S8 NEXTSECTOR S9
S9 NEXTSECTOR S29
S29 NEXTSECTOR S28
S28 NEXTSECTOR S20
S28 NEXTSECTOR S30
S20 NEXTSECTOR S27
S27 NEXTSECTOR S26
S26 NEXTSECTOR S4
S30 NEXTSECTOR S31
S31 NEXTSECTOR S32
S32 NEXTSECTOR S7
S9 NEXTSECTOR S10
S10 NEXTSECTOR S11
S11 NEXTSECTOR S12
S12 NEXTSECTOR S13
S14 NEXTSECTOR S15
S15 NEXTSECTOR S16
S16 NEXTSECTOR S17
S17 NEXTSECTOR S1
S1 NEXTSECTOR S2
S2 NEXTSECTOR S3
S3 NEXTSECTOR S4
// ————————.
```

Figure 31: Train : fichier des faits relatifs au train T1 (suite)

```
// ————————————————————
// Ce processeur est responsable du train T2
// ————————-
TRAIN NAME T2
T2 SECTOR S1
T2 POSITION 1
// ————————-
L1 STATE RED
L6 STATE RED
L4 STATE GREEN
L9 STATE GREEN
L7 STATE RED
L10 STATE RED
L11 STATE GREEN
L13 STATE GREEN
L15 STATE GREEN
L3 STATE GREEN
// ————————-
S1 LIGHTA L1
S1 LIGHTB L6
S2 LIGHTA L1
S2 LIGHTB L6
S3 LIGHTA L1
S3 LIGHTB L6
// ————————-
S4 LIGHTA L4
S4 LIGHTB L9
S5 LIGHTA L4
S5 LIGHTB L9
S6 LIGHTA L4
S6 LIGHTB L9
// ————————-
S7 LIGHTA L7
S7 LIGHTB L10
S8 LIGHTA L7
S8 LIGHTB L10
S9 LIGHTA L7
S9 LIGHTB L10
// ————————-
S10 LIGHTA L11
S10 LIGHTB L13
S11 LIGHTA L11
S11 LIGHTB L13
// ————————-
S12 LIGHTA L15
S12 LIGHTB L3
```

Figure 32: Train : fichier des faits relatifs au train T2

```
// ————————————————————
S13 LIGHTA L15
S13 LIGHTB L3
S14 LIGHTA L15
S14 LIGHTB L3
S15 LIGHTA L15
S15 LIGHTB L3
S16 LIGHTA L15
S16 LIGHTB L3
S17 LIGHTA L15
S17 LIGHTB L3
// ————————-
S1 VALUE 1
S2 VALUE 2
S3 VALUE 3
S4 VALUE 4
S5 VALUE 5
S6 VALUE 6
S7 VALUE 7
S8 VALUE 8
S9 VALUE 9
S10 VALUE 10
S11 VALUE 11
S12 VALUE 12
S13 VALUE 13
S14 VALUE 14
S15 VALUE 15
S16 VALUE 16
S17 VALUE 17
// ————————-
S1 NEXTSECTOR S2
S2 NEXTSECTOR S3
S3 NEXTSECTOR S4
S4 NEXTSECTOR S5
S5 NEXTSECTOR S6
S6 NEXTSECTOR S7
S7 NEXTSECTOR S8
S8 NEXTSECTOR S9
S9 NEXTSECTOR S10
S10 NEXTSECTOR S11
S11 NEXTSECTOR S12
S12 NEXTSECTOR S13
S13 NEXTSECTOR S14
S14 NEXTSECTOR S15
S15 NEXTSECTOR S16
S16 NEXTSECTOR S17
S17 NEXTSECTOR S1
// ————————-
```

Figure 33: Train : fichier des faits relatifs au train T2 (suite)

```
// ————————————————————
// Ce processeur est responsable du train T3
// ————————-    TRAIN NAME T3
T3 SECTOR S18
T3 POSITION 18
// ————————-
L2 STATE RED
L28 STATE RED
L26 STATE GREEN
L24 STATE GREEN
L23 STATE GREEN
L22 STATE GREEN
L20 STATE GREEN
L18 STATE GREEN
L16 STATE GREEN
L14 STATE GREEN
L27 STATE GREEN
L17 STATE GREEN
L15 STATE GREEN
L3 STATE GREEN
// ————————-
S18 LIGHTA L2
S18 LIGHTB L28
S19 LIGHTA L26
S19 LIGHTB L24
S20 LIGHTA L23
S20 LIGHTB L22
S21 LIGHTA L20
S21 LIGHTB L16
S22 LIGHTA L17
S22 LIGHTB L14
S23 LIGHTA L27
S23 LIGHTB L18
S24 LIGHTA L27
S24 LIGHTB L18
S25 LIGHTA L27
S25 LIGHTB L18
// ————————-
S12 LIGHTA L15
S12 LIGHTB L3
S13 LIGHTA L15
S13 LIGHTB L3
```

Figure 34: Train : fichier des faits relatifs au train T3

```
// ----------------------
S14 LIGHTA L15
S14 LIGHTB L3
S15 LIGHTA L15
S15 LIGHTB L3
S16 LIGHTA L15
S16 LIGHTB L3
S17 LIGHTA L15
S17 LIGHTB L3
// --------
S12 VALUE 12
S13 VALUE 13
S14 VALUE 14
S15 VALUE 15
S16 VALUE 16
S17 VALUE 17
S18 VALUE 18
S19 VALUE 19
S20 VALUE 20
S21 VALUE 21
S22 VALUE 22
S23 VALUE 23
S24 VALUE 24
S25 VALUE 25
// --------
S12 NEXTSECTOR S13
S13 NEXTSECTOR S14
S14 NEXTSECTOR S15
S15 NEXTSECTOR S16
S16 NEXTSECTOR S17
S17 NEXTSECTOR S18
S18 NEXTSECTOR S19
S18 NEXTSECTOR S23
S19 NEXTSECTOR S20
S20 NEXTSECTOR S21
S21 NEXTSECTOR S22
S23 NEXTSECTOR S24
S24 NEXTSECTOR S25
S25 NEXTSECTOR S22
S22 NEXTSECTOR S12
// --------
```

Figure 35: Train : fichier des faits relatifs au train T3 (suite)

```
// ————————————————————
// Ce processeur est responsable du train T4
// ————————-     TRAIN NAME T4
T4 SECTOR S8
T4 POSITION 8
// ————————-
L4 STATE GREEN
L9 STATE GREEN
L5 STATE GREEN
L8 STATE RED
L21 STATE RED
L7 STATE RED
L10 STATE RED
L12 STATE GREEN
L19 STATE GREEN
L22 STATE GREEN
L23 STATE GREEN
L25 STATE GREEN
// ————————-
S4 LIGHTA L9
S4 LIGHTB L4
S5 LIGHTA L9
S5 LIGHTB L4
S6 LIGHTA L9
S6 LIGHTB L4
// ————————-
S7 LIGHTA L10
S7 LIGHTB L7
S8 LIGHTA L10
S8 LIGHTB L7
S9 LIGHTA L10
S9 LIGHTB L7
// ————————-
S20 LIGHTA L23
S20 LIGHTB L22
// ————————-
S26 LIGHTA L5
S26 LIGHTB L25
S27 LIGHTA L5
S27 LIGHTB L25
// ————————-
S28 LIGHTA L12
S28 LIGHTB L19
S29 LIGHTA L12
S29 LIGHTB L19
```

Figure 36: Train : fichier des faits relatifs au train T4

```
// ----------
// -------
S30 LIGHTA L8
S30 LIGHTB L21
S31 LIGHTA L8
S31 LIGHTB L21
S32 LIGHTA L8
S32 LIGHTB L21
// -------
S4 VALUE 4
S5 VALUE 5
S6 VALUE 6
S7 VALUE 7
S8 VALUE 8
S9 VALUE 9
S20 VALUE 20
S26 VALUE 26
S27 VALUE 27
S28 VALUE 28
S29 VALUE 29
S30 VALUE 30
S31 VALUE 31
S32 VALUE 32
// -------
S4 NEXTSECTOR S26
S5 NEXTSECTOR S4
S6 NEXTSECTOR S5
S7 NEXTSECTOR S6
S8 NEXTSECTOR S7
S9 NEXTSECTOR S8
S29 NEXTSECTOR S9
S28 NEXTSECTOR S29
S20 NEXTSECTOR S28
S30 NEXTSECTOR S28
S31 NEXTSECTOR S30
S32 NEXTSECTOR S31
S7 NEXTSECTOR S32
S27 NEXTSECTOR S20
S26 NEXTSECTOR S27
// -------
```

Figure 37: Train : fichier des faits relatifs au train T4 (suite)

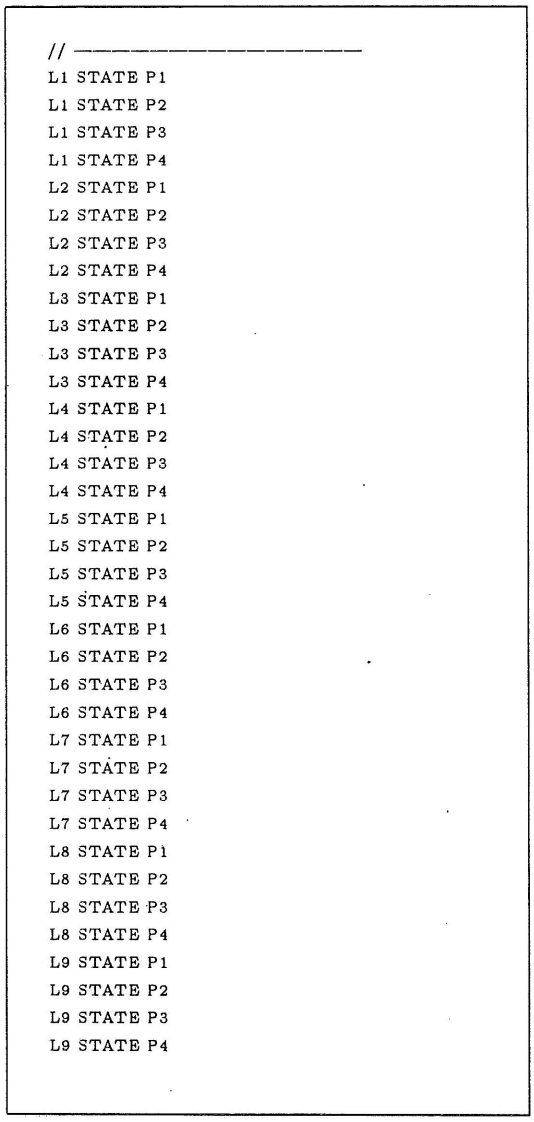


Figure 38: Train : fichier des routes pour ALEX

```
// ————————————————————
L10 STATE P1
L10 STATE P2
L10 STATE P3
L10 STATE P4
L11 STATE P1
L11 STATE P2
L11 STATE P3
L11 STATE P4
L12 STATE P1
L12 STATE P2
L12 STATE P3
L12 STATE P4
L13 STATE P1
L13 STATE P2
L13 STATE P3
L13 STATE P4
L14 STATE P1
L14 STATE P2
L14 STATE P3
L14 STATE P4
L15 STATE P1
L15 STATE P2
L15 STATE P3
L15 STATE P4
L16 STATE P1
L16 STATE P2
L16 STATE P3
L16 STATE P4
L17 STATE P1
L17 STATE P2
L17 STATE P3
L17 STATE P4
L18 STATE P1
L18 STATE P2
L18 STATE P3
L18 STATE P4
L19 STATE P1
L19 STATE P2
L19 STATE P3
L19 STATE P4
```

Figure 39: Train : fichier des routes pour ALEX (suite)

```
// ----------------------
L20 STATE P1
L20 STATE P2
L20 STATE P3
L20 STATE P4
L21 STATE P1
L21 STATE P2
L21 STATE P3
L21 STATE P4
L22 STATE P1
L22 STATE P2
L22 STATE P3
L22 STATE P4
L23 STATE P1
L23 STATE P2
L23 STATE P3
L23 STATE P4
L24 STATE P1
L24 STATE P2
L24 STATE P3
L24 STATE P4
L25 STATE P1
L25 STATE P2
L25 STATE P3
L25 STATE P4
L26 STATE P1
L26 STATE P2
L26 STATE P3
L26 STATE P4
L27 STATE P1
L27 STATE P2
L27 STATE P3
L27 STATE P4
L28 STATE P1
L28 STATE P2
L28 STATE P3
L28 STATE P4
```

Figure 40: Train : fichier des routes pour ALEX (suite)

```
// ————————————————————
L1 STATE P100
L2 STATE P100
L3 STATE P100
L4 STATE P100
L5 STATE P100
L6 STATE P100
L7 STATE P100
L8 STATE P100
L9 STATE P100
L10 STATE P100
L11 STATE P100
L12 STATE P100
L13 STATE P100
L14 STATE P100
L15 STATE P100
L16 STATE P100
L17 STATE P100
L18 STATE P100
L19 STATE P100
L20 STATE P100
L21 STATE P100
L22 STATE P100
L23 STATE P100
L24 STATE P100
L25 STATE P100
L26 STATE P100
L27 STATE P100
L28 STATE P100
T1 SECTOR P1
T1 POSITION P1
T2 SECTOR P2
T2 POSITION P2
T3 SECTOR P3
T3 POSITION P3
T4 SECTOR P4
T4 POSITION P4
T1 POSITION P100
T2 POSITION P100
T3 POSITION P100
T4 POSITION P100
```

Figure 41: Train : fichier des routes pour ALEX (suite)

Annexe D: Le chemin de fer : les images de l'interface

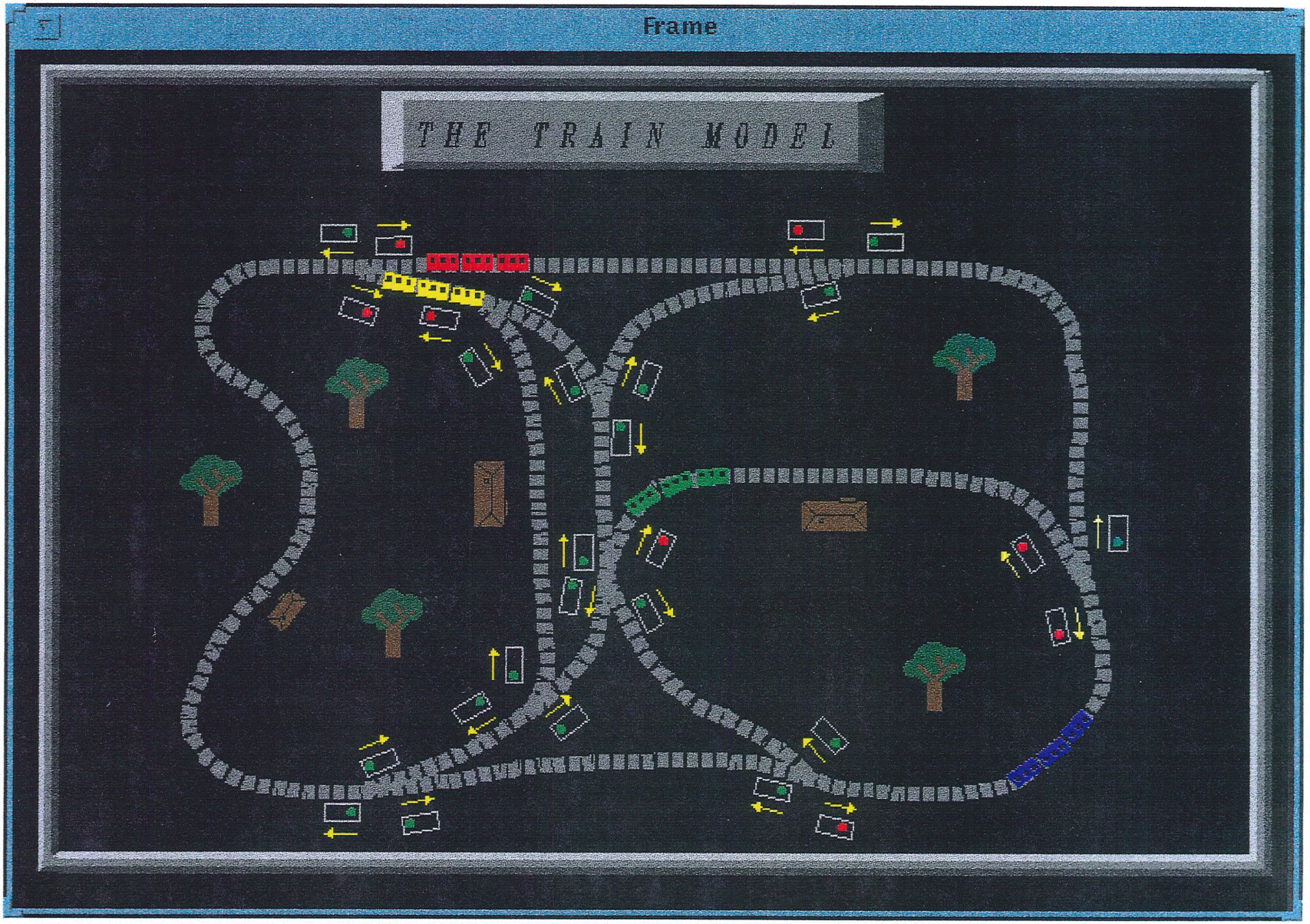
Les quatre pages qui suivent présentent le problème de la mine à différents stades d'exécution.

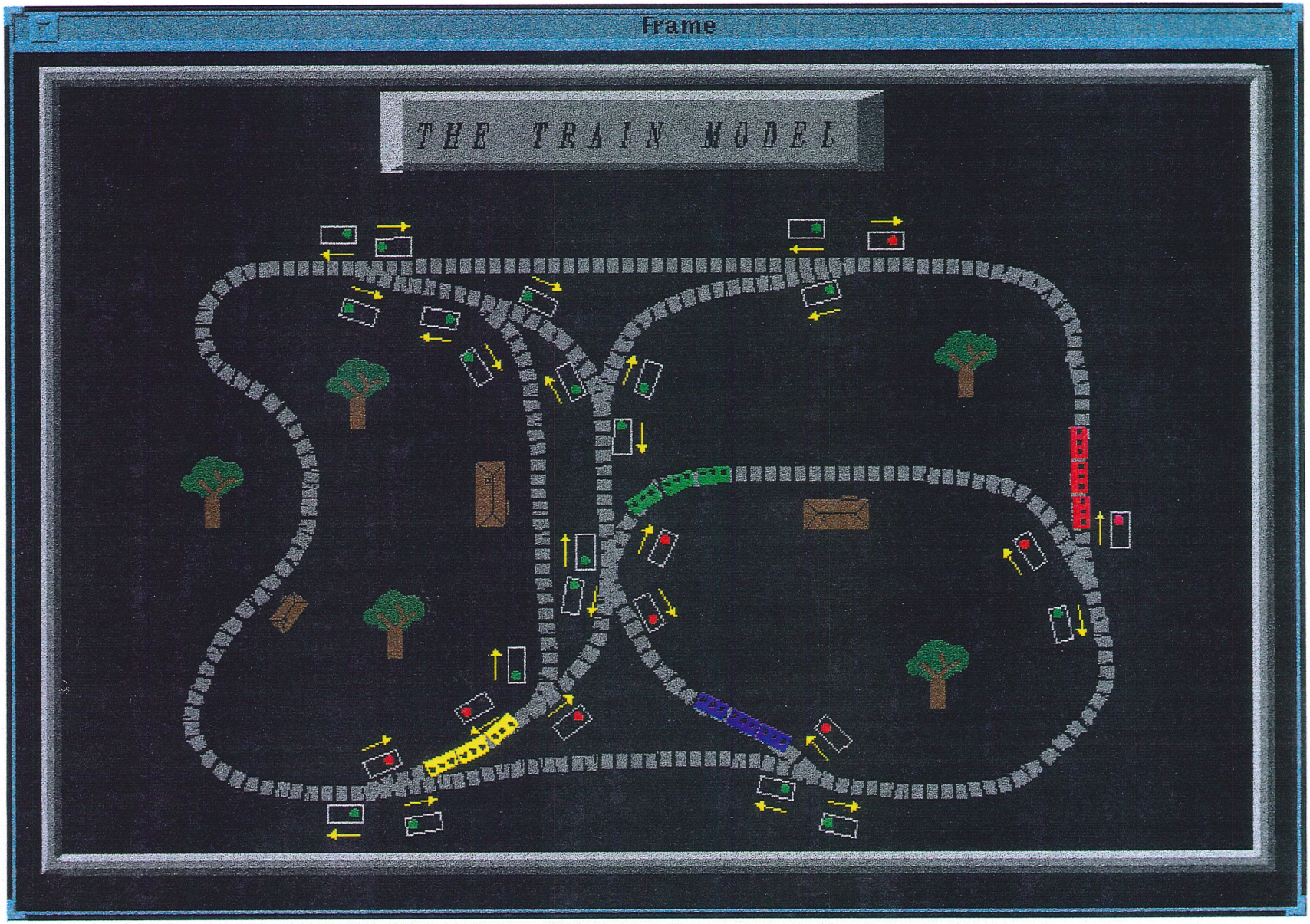
Première image : Cette image représente l'état initial du problème. Les trains bleu (en bas à droite) et vert (au milieu) circulent dans le sens contraire des aiguilles d'une montre. Le train jaune et le train rouge circulent dans le sens des aiguilles d'une montre. Le train jaune est limité par son fichier de routage et ne peut circuler que sur les deux cercles de la partie gauche du modèle. Le train rouge effectue le grand cercle.

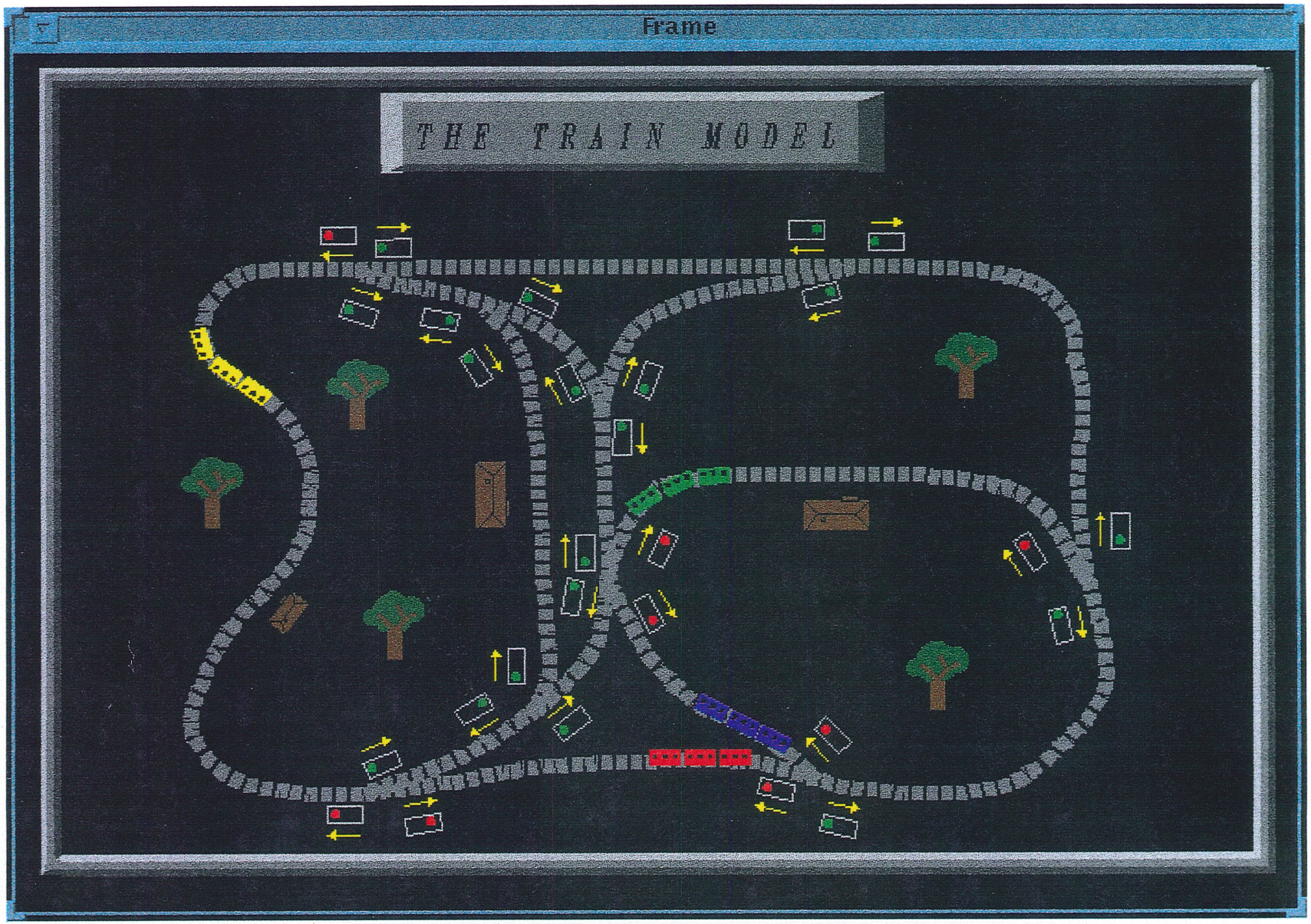
Seconde image : Cette image nous montre l'état du modèle après quelques inférences.

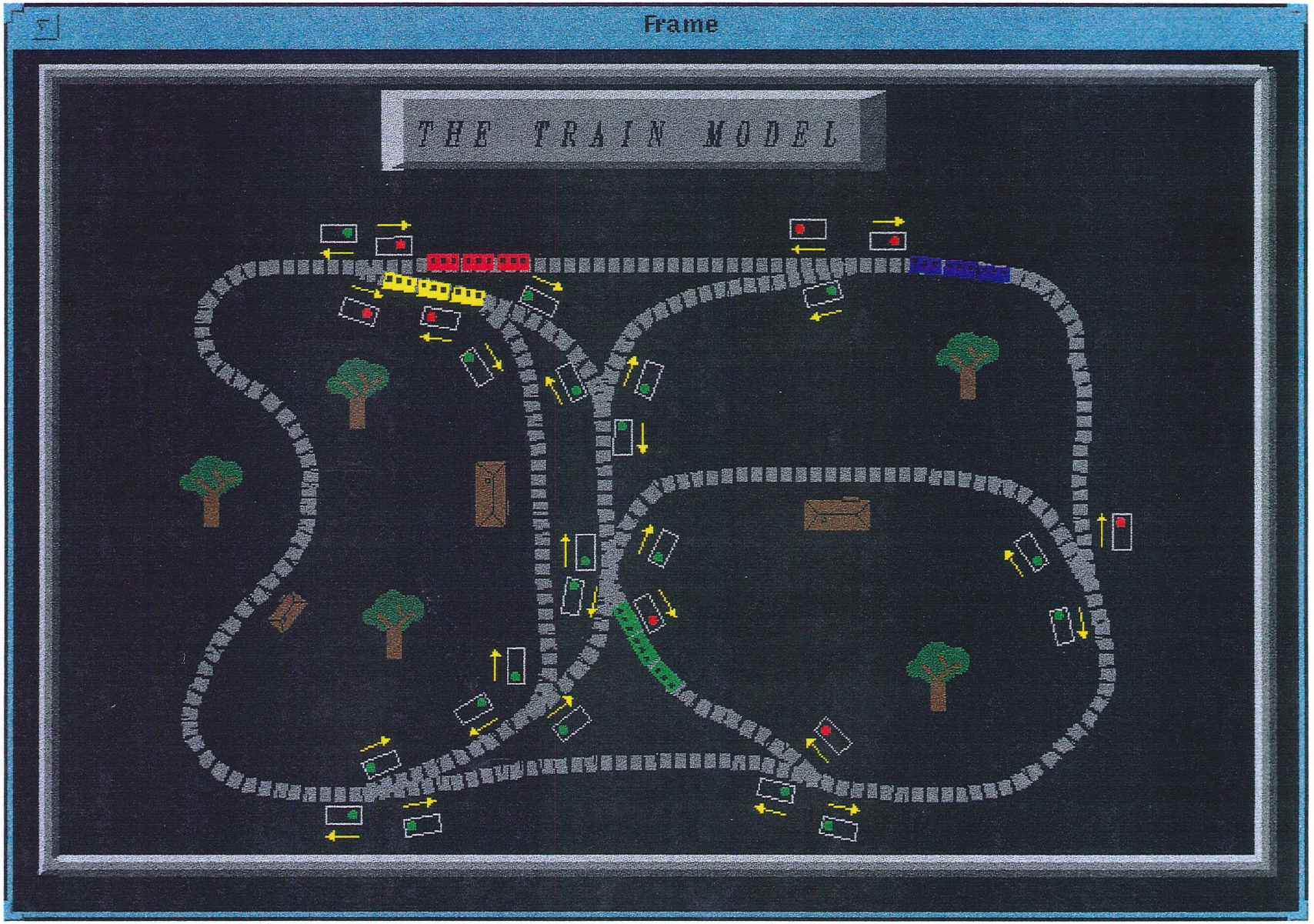
Troisième image : Cette image nous montre l'état du modèle après plusieurs inférences.

Quatriéme image : Sur cette image, le train vert s'est mis à bouger. Le train jaune est revenu à son état initial.


Frame
THE TRAIN MODEL


Frame
THE TRAIN MODEL


Frame
THE TRAIN MODEL


Frame
THE TRAIN MODEL

Annexe E: Syntaxe

Cette annexe présente la grammaire, sous forme Backus-Naur, des différents fichiers de paramètres. Tous les mots en lettres majuscules sont des mots-clés et aucune règle ne peut les utiliser hors de leur contexte. Les noms des fonctions doivent être rigoureusement les mêmes.

```
<fichier_regles> := {<regle> | <commentaire>}*

<commentaire> := "// " {<id>}*

<regle>          := <nom_regle> <priorite_statique>
                        "IF"
                        <antecédent> { <antecedent>}*
                    "THEN"
                    <consequent> { <consequent>}*
                    "FI"

<nom_regle>        := <id>

<priorite_statique>  := "PS"<digit>

<antecedent>       := {<fait> | <fonction>}*

<consequent>      := {<mutateur> <fait> | <fonction>}*
```

```
<mutateur>  := "INSERT" | "DELETE" | "ADDTO" | "SUBFROM" | "REPLACE"

<fait>           := <objet> <attribut> <valeur>

<fonction>  := <function_name> <parametre_entree> <parametre_entree>
                            <parametre_sortie>

<nom_fonction> := "Plus" | "Minus" | "Mult" | "Greater" | "Smaller" | "Equal"

<parametre_entree> := <constante> | <variable>

<parametre_sortie> := <variable> | "$null"

<objet>          := <id> | <variable>

<attribut>      := <id> | <variable>

<valeur>           := <id> | <variable> | <entier>

<constante>       := <id>

<variable>        := $<id>

<id> := <lettre>{<lettre>|<digit>}*

<fichier_fait> := {<instance>} | {<commentaire>}*

<instance> := {<i_objet> <i_attribut> <i_valeur>}*

<i_objet>          := <id>

<i_attribut>      := <id>
```

```
<i_valeur>          := <id> | <entier>

<fichier_routage> := {<i_objet> <i_attribute> <processor>} | {<commentaire>}*

<processeur> := "P1" | "P2" | "P3" | "P4" | "P5" | "P6" | "P7" | "P8" |
                "P9" | "P10" | "P11" | "P12" | "P13" | "P14" | "P15" | "P16" | "P100"

<fichier_nom> := "OBJETS" {<id>}* "FIN_TABLEAU"
      "ATTRIBUTS" {<id>}* "FIN_TABLEAU"
       "FONCTIONS" {<id>}* "FIN_TABLEAU"
       "VARIABLES" {<id>}* "FIN_TABLEAU"
       "CONSTANTES" {<id>}* "FIN_TABLEAU"
       "VAL_CONSTANTES" {<digit>}* "FIN_TABLEAU"
       "REGLES" {<id>}* "FIN_TABLEAU"

<commentaire>           := "// " {<id>}*
```

Bibliographie

[Ale92a] ALEX INFORMATIQUE INC., *AVX Series 2 Reference Manuals*, ALEX INFORMATIQUE INC., 1992.

[BAR94] M. BARBEAU, G. CUSTEAU ET R. ST-DENIS, *Spécification des besoins et synthèse d'un contrôleur*, RAIRO, AUTOMATIQUE PRODUCTIQUE INFORMATIQUE INDUSTRIELLE, VOLUME 28(1), P. 37-52, 1994.

[BOO94] G. BOOCH, *Object-oriented analisys and design*, THE BENJAMIN/CUMMINGS PUBLISHING COMPAGNY INC., 1994.

[BRO91] A. BROGI, A.. CIAMPOLINI, E. LAMMA, P. MELLO *A distributed implementation for parallel logic programming*, PROCEEDINGS OF THE 5TH EUROPEAN COMPUTER CONFERENCE ON ADVANCED COMPUTER TECHNOLOGY, RELIABLE SYSTEMS AND APPLICATION - COMPEURO 91 MAY 13-16 ITALY, P. 118-122, 1991.

[CHO85] E. CHOURAQUI ET AL., *Modélisation du raisonnement et de la connaissance*, TECHNIQUE ET SCIENCE INFORMATIQUE VOL. 4 NO. 4, P. 391-399, 1985.

[COR81] FRANÇOISE ANDRÉ ET AL., *Systèmes informatiques répartis*, BORDAS, PARIS, 1981.

[LAR94] *Petit Larousse Illustré*, EDITION LAROUSSE, 1994.

[PAL95] J. M. PALMIER, *Réalisation d'interfaces graphiques pour un contrôleur de procédés industriels*, UNIVERSITÉ DE SHERBROOKE, 1993.

[STD94] R. ST-DENIS, *Notes du cours de deuxième cycle IFT-724*, UNIVERSITÉ DE SHERBROOKE, 1994.

[VIR93] VIRTUAL PROTOTYPES INC., *VAPS (3.0) Reference Guide*, VIRTUAL PROTOTYPES INC., 1993.